OTR version 4

> Disclaimer
>
> This protocol specification is a draft. It's currently under constant revision.

This document describes version 4 of the Off-the-Record Messaging protocol. OTR version 4 (OTRv4) provides better deniability properties by the use of a deniable authenticated key exchange (DAKE), and better forward secrecy through the use of double ratcheting. OTRv4 works on top of an existing messaging protocol, such as XMPP.

Table of Contents

Main Changes over Version 3

- Security level raised to 224 bits and based on Elliptic Curve Cryptography (ECC).
- Additional protection against transcript decryption in the case of ECC compromise.
- Support of conversations where one party is offline.
- Updated cryptographic primitives and protocols:
    - Deniable authenticated key exchanges (DAKE) using "DAKE with Zero Knowledge" (DAKEZ) and "Extended Zero-knowledge Diffie-Hellman" (XZDH) [1]. DAKEZ corresponds to conversations when both parties are online (interactive) and XZDH to conversations when one of the parties is offline (non-interactive).
    - Key management using the Double Ratchet Algorithm [2].
    - Upgraded SHA-1 and SHA-2 to SHAKE-256.
    - Switched from AES to ChaCha20 [3]. The RFC 7539 variant is used [16] .
- Support of an out-of-order network model.
- Support of different modes in which this specification can be implemented.
- Explicit instructions for producing forged transcripts using the same functions used to conduct honest conversations.

Reasons for the decisions made above and more are included in the architectural decisions records.

High Level Overview

An OTRv4 conversation may begin when the two participants are online (an interactive conversation) or when one participant is offline (non-interactive conversation).

Conversation started by an Interactive DAKE

```
Alice                                   Bob
------------------------------------------------------------------------------
Requests OTR conversation               ------------>

Establishes Conversation with DAKEZ  <------------>  Establishes Conversation with DAKEZ

Exchanges Data Messages              <------------>  Exchanges Data Messages
```

The conversation can begin after one participant requests a conversation. This includes an advertisement of which versions the participant supports. If the other participant supports OTRv4, an interactive DAKE can be used to establish a secure channel. Encrypted messages are then exchanged in this secure channel with strong forward secrecy.

Conversation started by a Non-Interactive DAKE

```
Alice                       Untrusted Prekey Server    Bob
------------------------------------------------------------------------------
                          (<----------------------   Pre-conversation: Creates
                                                     and sends Prekey Ensembles: creates
                                                     a Client Profile, a Prekey Profile and a set of
                                                     prekey messages)
Retrieves Bob's  ----------------->
Prekey Ensemble: asks for
a Client Profile, a Prekey
Profile and a prekey message

Establishes Conversation  ------------------------------->
with XZDH and sends the
first Data Message
```

Exchanges Data Messages <--------------------------------> Exchanges Data Messages

The conversation can begin when one participant retrieves the other's participant Prekey Ensemble from an untrusted Prekey Server (consisting of a Client Profile, a Prekey Profile and a set of prekey messages). Prior to the start of the conversation, these Prekey values would have had to be uploaded by the other participant's client to a server. This have to be done so other participants, like Alice, can send messages to the other participant, like Bob, while the latter is offline.

Definitions

Unless otherwise noted, these conventions and definitions are used for this document:

- "Adversary" refers to a malicious entity whose aim is to prevent the participants of this protocol from achieving their goal.
- "Client" refers to a software used for communication between parties. It implements the OTRv4 protocol.
- "OTR Conversation" and "Conversation" refers to an interaction between parties by exchanging encrypted messages with each other using the OTRv4 protocol.
- "DAKE" refers to a Deniable Authenticated Key Exchange, in which two parties engage in the protocol whose result is a key which only the two of them know, and they are assured to be sharing it with each other. They will use it to encrypt and authenticate messages in the session, using an authentication mechanism that is deniable provided that the key cannot be traced to either party.
- "Informant" refers to an insider with privileged access capable of divulging information.
- "Initiator" refers to the participant initiating a DAKE. In the case of the interactive DAKE, this is the participant that sends the Identity message. In the case of the non-interactive DAKE, this is the participant that uploads a Prekey Ensemble.
- "Judge" refers to an entity that decide whether or not a certain event occurred. We say that an action is deniable with respect to a given judge if the judge cannot be convinced that an individual performed the action.
- "Mode" refers to a way in which OTRv4 can be implemented.
- "Network" refers to the system that computing devices use to exchange data with each other using connections between nodes.
- "Participant" or "Party" refers to any of the end-points that take part in a conversation.
- "Prekey Server" refers to the untrusted server used to store Prekey Ensembles.
- "Publisher" refers to the participant publishing Prekey Ensembles to the Prekey Server.
- "Receiver" refers to the participant receiving an encoded message.
- "Responder" refers to the participant responding to an Initiator's request. In the case of the interactive DAKE, this is the participant that sends the Auth-R message. In the case of the non-interactive DAKE, this is the participant that sends the Non-Interactive Auth message.
- "Retriever" refers to the participant retrieving Prekey Ensembles from the Prekey Server that correspond to the Publisher.
- "Sender" refers to the participant sending an encoded message.
- "Session" refers to a semi-permanent information interchange between parties. It is not only limited to the exchange of encrypted messages.

Assumptions

Messages in a conversation can be exchanged over an insecure channel, where an attacker can eavesdrop or interfere with the messages.

The network model provides in-order and out-of-order delivery of messages. Some messages may not be delivered.

OTRv4 does not protect against an active attacker performing Denial of Service attacks.

Security Properties

OTRv4 is the version 4 of the cryptographic protocol OTR. It provides end-to-end encryption, which is a system by which information is sent over a network in such a way that only the recipient and sender can read it.

OTRv4 provides trust establishment (user verification) by fingerprint verification or by the ability to perform the Socialist Millionaires Protocol (SMP). The latter is a zero-knowledge proof of knowledge protocol that determines if secret values held by two parties are equal without revealing the value itself.

OTRv4 uses two kinds of Deniable Authenticated Key Exchanges (DAKE), as stated above: one for interactive conversations and one for non-interactive conversations.

In the interactive DAKE, although access to one participant's private long-term key is required for authentication, both participants can deny having used their private long-term keys. A forged transcript of the DAKE can be produced by anyone who knows the long-term public keys of both alleged participants. This capability is called offline deniability because no transcript provides evidence of a past key exchange, as it could have being forged by anyone. This property is provided for both participants taking part in the interactive DAKE as described below.

Furthermore, participants in the interactive DAKE, cannot provide proof of participation to third parties without making themselves vulnerable to Key Compromise Impersonation (KCI) attacks [11], even if they perform arbitrary protocols with these third parties. A KCI attack begins when the long-term secret key of a participant of a vulnerable DAKE is compromised. With this secret key, an adversary can impersonate other users to the owner of the key. The property by which participants cannot provide proof of participation to third parties is known as online deniability. Notice that OTRv4 uses 'forging keys' to provide online deniability and to prevent KCI attacks at the same time. For a detailed explanation around how forging keys work in regards to online deniability and KCI attacks, refer to section KCI attacks.

Online deniability can be broken in two ways: 1. coercive judges, when an online judge coerces a participant into interactively proving that messages were authored by a victim, without compromising long-term secrets; 2. malicious users, when a malicious participant interacts with a purpose-built third-party service during a conversation with a victim to produce non-repudiable proof of message authorship by the victim. This second attack can happen with the help of remote attestation, where an adversary uses it on a participant's client to produce a non-repudiable proof/transcript of the otherwise deniable protocol [12].

Both DAKEs (interactive and non-interactive) provide offline deniability as anyone can forge a DAKE transcript between two parties using their long-term public keys. The interactive DAKE provides online deniability for both parties.

In the non-interactive DAKE, the initiator (Bob, in the above overview) has online deniability, but Alice, the responder, does not. This happens as there can exist a protocol whereby a third party, with Alice's help, can establish an authenticated conversation with Bob in Alice's name without having to learn her private keys. This generates irrefutable cryptographic proof that a conversation took place.

Although both DAKEs (interactive and non-interactive) provide deniability, take into account that there may be a loss of deniability if an interactive DAKE is followed by a non-interactive one.

Once a conversation has been established with the DAKE, all data messages transmitted in it are confidential and retain their integrity. They are authenticated using a MAC. As MAC keys are published and OTRv4 uses malleable encryption, anyone can forge data messages, and consequently, deny their contents.

Furthermore, OTRv4 provides forward secrecy. An adversary that compromises the long-term secret keys of both parties cannot retroactively compromise past session keys. The interactive DAKE offers strong forward secrecy (it protects the session key when at least one party completes the exchange). The non-interactive DAKE offers a forward secrecy that is between strong and weak, as it protects completed sessions and incomplete sessions that stall long enough to be invalidated by a participant. The key exchange mechanism used in OTRv4 is the Double Ratchet algorithm which provides forward and backward secrecy, as parties negotiate secrets several times using an ephemeral key exchange.

A protocol provides forward secrecy if the compromise of a long-term key does not allow ciphertexts encrypted with previous session keys to be decrypted. If the compromise of a long-term key does not allow subsequent ciphertexts to be decrypted by passive attackers, then the protocol is said to have backward secrecy. Furthermore, if the compromise of a single session key is not permanent, as, after some time, subsequent messages will be impossible to decrypt again because of the "self-healing" nature of the algorithm, then the protocol is said to have post-compromise security. In the current academic literature, backwards secrecy is included in post-compromise security. OTRv4, by using the Double Ratchet Algorithm, provides these two properties: forward secrecy and post-compromise security. If key material used to encrypt a particular data message is compromised, previous and future messages are protected.

The DAKEs in OTRv4 provide contributiveness as well. This means that the initiator of the protocol cannot force the shared secret to take on a specific value. It is also computationally infeasible for the responder to select a specific shared secret. Additionally, both DAKEs in this specification are provable secure, meaning that both of them come with a rigorous logical argument as proof.

OTRv4 does not take advantage of quantum resistant algorithms. There are several reasons for this. Mainly, OTRv4 aims to be a protocol that is easy to implement in today's environments and within a year. Current quantum resistant algorithms and their respective implementations are not ready enough to allow for this implementation time frame. As a result, the properties mentioned in these paragraphs only apply to non-quantum adversaries.

The only exception is the usage of a "brace key" to provide some post-conversation transcript protection against potential weaknesses of elliptic curves and the early arrival of quantum computers. Nevertheless, notice that the "brace key" does not provide any kind of post-quantum confidentiality. When fault-tolerant quantum computers break Ed448-Goldilocks keys, it will take some years beyond that point to break 3072-bit Diffie-Hellman keys.

These security properties only hold for when a conversation with OTRv4 is started. They do not hold for the previous versions of the OTR protocol, meaning that if a user that supports version 3 and 4 starts a conversation with someone that only supports version 3, a conversation with OTRv3 will start, and its security properties will not be the ones stated in these paragraphs.

OTRv4 Modes

In order for OTRv4 to be an alternative to current messaging applications, to be compatible with the OTRv3 specification and to be useful for instant messaging protocols (e.g. XMPP), the OTRv4 protocol must define different modes in which it can be implemented: a OTRv3-compatible mode, a OTRv4 standalone mode, and a OTRv4 interactive-only-mode. These are the three modes enforced by this protocol

specification; but, it must be taken into account, that OTRv4 can and may be also implemented in other modes.

The modes are:

1. OTRv3-compatible mode: a mode with backwards compatibility with OTRv3. This mode will know how to handle plaintext messages, including query messages and whitespace tags.
2. OTRv4-standalone mode: an always encrypted mode. This mode will not know how to handle any kind of plaintext messages, including query messages and whitespace tags. It supports both interactive and non-interactive conversations. It is not backwards compatible with OTRv3.
3. OTRv4-interactive-only: an always encrypted mode that provides higher deniability properties when compared to the previous two modes, as it achieves offline and online deniability for both participants in a conversation. It only supports interactive conversations. It is not backwards compatible with OTRv3. This mode can be used by network models that do not have a central infrastructure, like Ricochet (keep in mind, though, that if OTRv4 is used over Ricochet, some online deniability properties will be lost).

For details on how these modes work, and how the DAKEs and double ratchet is initialized in them, review the modes folder.

Take into account, that some clients might implement different modes when talking with each other. In those cases:

- If a client implements "OTRv4-standalone" mode or "OTRv4-interactive-only" mode and a request for an OTRv3 conversation arrives, reject this request.
- If a client implements "OTRv4-interactive-only" mode and a request for an offline conversation arrives, reject this request.

The OTRv4 state machine will also need to know the mode in which is working on when initialized. It will also need to take this mode into account every time it makes a decision around how to transition from every state.

Notation and Parameters

This section contains information needed to understand the parameters, variables and arithmetic used in the specification.

Notation

Scalars and secret keys are in lower case, such as x or y. Points and public keys are in upper case, such as P or Q.

Addition of elliptic curve points A and B is $A + B$. Subtraction is $A - B$. Addition of a point to another point generates a third point. Scalar multiplication of an elliptic curve point B with a scalar a yields a new point: $C = B * a$. For details on how to implement these operations, see the Elliptic Curve Operations section.

The concatenation of byte sequences I and J is $I \mid\mid J$. In this case, I and J represent a fixed-length byte sequence encoding of the respective values. See the section on Data Types for encoding and decoding details.

A scalar modulo q is a field element, and should be encoded and decoded as a SCALAR type, which is defined in the Data Types section.

A point should be encoded and decoded as a POINT type, which is defined in the Data Types section.

The byte representation of a value x is defined as $\text{byte}(x)$.

The endianness is little and big-endian. Data types that are specific to elliptic curve arithmetic (POINT, SCALAR, ED448-PUBKEY, $\text{ED448-SHARED-PREKEY}$ and EDDSA-SIG) are encoded as little-endian. The rest of data types are encoded as big-endian. Little-endian encoding into bits places bits from left to right and from least significant to most significant. Big-endian encoding into bits places bits from right to left and from most significant to least significant.

Elliptic Curve Parameters

OTRv4 uses the Ed448-Goldilocks [4] elliptic curve [5]. Ed448-Goldilocks is an untwisted Edwards curve, where:

```
Equation
  x^2 + y^2 = 1 - 39081 * x^2 * y^2

Coordinates:
  Affine coordinates

Base point (G)
  (x=224580040295924300187604334099896036246789641632564134246125461686950415467406032909029192869357953282578032075146446173674602635247710,
   y=298819210078481492676017930443930673437544040154080242095928241372331506189835876003536878655418784733982303233503462500531545062832660)

Cofactor (c)
  4
```

```
Identity element (I)
  (x=0,
   y=1)

Field prime (p)
  2^448 - 2^224 - 1

Order of base point (q) [prime; q < p; q * G = I]
  2^446 - 138180668098951153520073867485154268803366924748821786098945475038 85

Non-square element in Z_p (d)
  -39081
```

Verifying that a point is on the curve

To verify that a point (`X = x, y`) is on curve Ed448-Goldilocks [14]:

1. Check that `X` is not equal to the identity element (`I`).
2. Check that `X` lies on the curve: this can be done by checking that `x` and `y` are integers on the interval `[0, p - 1]`.
3. Check that `q * X = I`.

Considerations while working with elliptic curve parameters

For any 57 bytes random value chosen in `Z_q` used for an elliptic curve operation (denoted `value`), hash it directly into a 57-byte large buffer `h` by doing `h = SHAKE-256(value, 57)`.

To prevent small subgroup attacks, prune the buffer:

The two least significant bits of the first byte are cleared, all eight bits of the last byte are cleared, and the highest bit of the second to last byte is set.

Interpret this pruned buffer as the little-endian integer, forming a secret scalar.

Take into account these operations when choosing random values for the Socialist Millionaires Protocol and the Ring Signature of Authentication.

3072-bit Diffie-Hellman Parameters

For the Diffie-Hellman (DH) group computations, the group is the one defined in RFC 3526 [6] with a 3072-bit modulus (hex, big-endian):

```
Prime (dh_p):
  2^3072 - 2^3008 - 1 + 2^64 * (integer_part_of(2^2942 * π) + 1690314)

Hexadecimal value of dh_p:
  FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
  29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
  EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
  E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
  EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
  C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
  83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
  670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
  E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
  DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
  15728E5A 8AAAC42D AD33170D 04507A33 A85521AB DF1CBA64
  ECFB8504 58DBEF0A 8AEA7157 5D060C7D B3970F85 A6E1E4C7
  ABF5AE8C DB0933D7 1E8C94E0 4A25619D CEE3D226 1AD2EE6B
  F12FFA06 D98A0864 D8760273 3EC86A64 521F2B18 177B200C
  BBE11757 7A615D6C 770988C0 BAD946E2 08E24FA0 74E5AB31
  43DB5BFC E0FD108E 4B82D120 A93AD2CA FFFFFFFF FFFFFFFF

Generator (g3)
  2

Cofactor
  2

Subprime (dh_q):
  (dh_p - 1) / 2

Hexadecimal value of dh_q:
  7FFFFFFF FFFFFFFF E487ED51 10B4611A 62633145 C06E0E68
  94812704 4533E63A 0105DF53 1D89CD91 28A5043C C71A026E
  F7CA8CD9 E69D218D 98158536 F92F8A1B A7F09AB6 B6A8E122
```

```
F242DABB 312F3F63 7A262174 D31BF6B5 85FFAE5B 7A035BF6
F71C35FD AD44CFD2 D74F9208 BE258FF3 24943328 F6722D9E
E1003E5C 50B1DF82 CC6D241B 0E2AE9CD 348B1FD4 7E9267AF
C1B2AE91 EE51D6CB 0E3179AB 1042A95D CF6A9483 B84B4B36
B3861AA7 255E4C02 78BA3604 650C10BE 19482F23 171B671D
F1CF3B96 0C074301 CD93C1D1 7603D147 DAE2AEF8 37A62964
EF15E5FB 4AAC0B8C 1CCAA4BE 754AB572 8AE9130C 4C7D0288
0AB9472D 45556216 D6998B86 82283D19 D42A90D5 EF8E5D32
767DC282 2C6DF785 457538AB AE83063E D9CB87C2 D370F263
D5FAD746 6D8499EB 8F464A70 2512B0CE E771E913 0D697735
F897FD03 6CC50432 6C3B0139 9F643532 290F958C 0BBD9006
5DF08BAB BD30AEB6 3B84C460 5D6CA371 047127D0 3A72D598
A1EDADFE 707E8847 25C16890 549D6965 7FFFFFFF FFFFFFFF
```

Whenever you see an operation on a field element from this group, the operation should be done modulo the prime `dh_p`.

Verifying that an integer is in the DH group

To verify that an integer (`x`) is on the group with a 3072-bit modulus:

1. Check that `x` is `>= g3` and `<= dh_p - g3`.
2. Compute `x ^ dh_q mod dh_p`. If `result == 1`, the integer is a valid element. Otherwise the integer is an invalid element.

Key Derivation Function, Hash Function and MAC Function

The following functions are used:

```
KDF(usage_ID || values, size) = SHAKE-256("OTRv4" || usage_ID || values, size)
HWC(usage_ID || values, size) = SHAKE-256("OTRv4" || usage_ID || values, size)
HCMAC(usage_ID || values, size) = SHAKE-256("OTRv4" || usage_ID || values, size)
```

The Key Derivation Function (KDF) is a function used when generating a key. The Hash with Context Function (HWC) is a function used to generate a hash. The Hash with Context Message Authentication Code is a function used to generate a MAC.

The `size` first bytes of the SHAKE-256 output for input `"OTRv4" || usage_ID || m` are returned for the three functions. Unlike SHAKE standard, notice that the output size here is defined in bytes.

The only different KDF function used in this specification is the one used when referring to RFC 8032. As defined in that document:

```
SHAKE-256(x, y) = The 'y' first bytes of SHAKE-256 output for input 'x'
```

The following usageID variables are defined:

```
* usage_fingerprint = 0x00
* usage_third_brace_key = 0x01
* usage_brace_key = 0x02
* usage_shared_secret = 0x03
* usage_SSID = 0x04
* usage_auth_r_bob_client_profile = 0x05
* usage_auth_r_alice_client_profile = 0x06
* usage_auth_r_phi = 0x07
* usage_auth_i_bob_client_profile = 0x08
* usage_auth_i_alice_client_profile = 0x09
* usage_auth_i_phi = 0x0A
* usage_first_root_key = 0x0B
* usage_tmp_key = 0x0C
* usage_auth_MAC_key = 0x0D
* usage_non_int_auth_bob_client_profile = 0x0E
* usage_non_int_auth_alice_client_profile = 0x0F
* usage_non_int_auth_phi = 0x10
* usage_auth_MAC = 0x11
* usage_root_key = 0x12
* usage_chain_key = 0x13
* usage_next_chain_key = 0x14
* usage_message_key = 0x15
* usage_MAC_key = 0x16
* usage_extra_symm_key = 0x17
* usage_authenticator = 0x18
* usage_SMP_secret = 0x19
* usage_auth = 0x1A
```

Data Types

OTRv4 uses many of the data types already specified in OTRv3 specification:

```
Bytes (BYTE):
  1 byte unsigned value
Shorts (SHORT):
  2 byte unsigned value, big-endian
Ints (INT):
  4 byte unsigned value, big-endian
Multi-precision integers (MPI):
  4 byte unsigned len, big-endian
  len byte unsigned value, big-endian
  (MPIs must use the minimum-length encoding; i.e. no leading 0x00 bytes.
   This is important when calculating public key fingerprints)
Opaque variable-length data (DATA):
  4 byte unsigned len, big-endian
  len byte data
```

OTRv4 also uses the following data types:

```
Nonce (NONCE):
  12 bytes data

Message Authentication Code (MAC):
  64 bytes MAC data

Ed448 point (POINT):
  57 bytes as defined in "Encoding and Decoding" section, little-endian

Ed448 scalar (SCALAR):
  57 bytes as defined in "Encoding and Decoding" section, little-endian

Client Profile (CLIENT-PROF):
  Detailed in "Client Profile Data Type" section

Prekey Profile (PREKEY-PROF)
  Detailed in "Prekey Profile Data Type" section
```

In order to encode a point or a scalar into POINT or SCALAR data types, and to decode a POINT or SCALAR data types into a point or a scalar, refer to the Encoding and Decoding section.

Encoding and Decoding

This section describes the encoding and decoding schemes specified in RFC 8032 [9] for scalars and points. Note that, although the RFC 8032 defines parameters as octet strings, they are defined as bytes here. It also describes the encoding of the OTRv4 messages that should be transmitted encoded.

Scalar

Encoded as a little-endian array of 57 bytes, e.g. $h[0] + 2^8 * h[1] + ... + 2^{447} * h[56]$. Take into account that scalars used for public key generation are not sent over the wire. Any random value chosen in Z_q that is going to be encoded, should have been hashed and pruned as defined in the Considerations while working with elliptic curve parameters section.

It is decoded to by interpreting a byte array (buffer) as an unsigned value, little-endian, and doing $\bmod q$ with the result. Note that every time a $\bmod q$ is specified for decoding, it is referred to this decoding.

Point

A curve point (x,y), with coordinates in the range $0 <= x,y < p$, is encoded as follows:

1. Encode the y-coordinate as a little-endian array of 57 bytes. The final byte is always zero.
2. Copy the least significant bit of the x-coordinate to the most significant bit of the final byte. This is 1 if the x-coordinate is negative or 0 if it is not.

A curve point is decoded as follows:

1. Interpret the 57-byte array as an integer in little-endian representation.
2. Interpret bit 455 as the least significant bit of the x-coordinate. Denote this value x_0. The y-coordinate is recovered simply by clearing this bit. If the resulting value is $>= p$, decoding fails.

3. To recover the x-coordinate, the curve equation implies `x^2 = (y^2 - 1) / (d * y^2 - 1) (mod p)`. The denominator is always non-zero mod `p`.

    1. Let `num = y^2 - 1` and `denom = d * y^2 - 1`. To compute the square root of `(num/denom)`, compute the candidate root `x = (num/denom)^((p+1)/4)`. This can be done using a single modular powering for both the inversion of `denom` and the square root:

    ```
    x = ((num ^ 3) * denom * (num^5 * denom^3) ^ ((p-3)/4)) (mod p)
    ```

    2. If `denom * x^2 = num`, the recovered x-coordinate is `x`. Otherwise, no square root exists, and decoding fails.

4. Use the `x_0` bit to select the right square root:
    - If `x = 0`, and `x_0 = 1`:
        - Decoding fails.
    - Otherwise, if `x_0 != x mod 2`:
        - Set `x <-- p - x`.
        - Return the decoded point `(x,y)`.

Encoded Messages

OTRv4 messages must be base-64 encoded. To transmit one of these messages, construct an ASCII string: the five bytes "?OTR:", the base-64 encoding of the binary form of the message and the byte ".".

Serializing the Ring Signature Proof of Authentication

The Ring Signature's non-interactive zero-knowledge proof of authentication is serialized as follows:

```
Ring Signature Authentication (RING-SIG):
  c1 (SCALAR)
  r1 (SCALAR)
  c2 (SCALAR)
  r2 (SCALAR)
  c3 (SCALAR)
  r3 (SCALAR)
```

Public keys, Shared Prekeys, Forging keys and Fingerprints

OTR users have long-lived public keys that they use for authentication (but not for encryption). OTRv4 introduces a new type of this long-term public key:

```
OTRv4 public authentication Ed448 key (ED448-PUBKEY):

  Pubkey type
    2 byte unsigned value, little-endian
    Ed448 public keys have type 0x0010

  H (POINT)
    H is the Ed448 public key generated as defined in RFC 8032.
```

In addition, OTRv4 also has a long-lived forging key, used for online deniability purposes and to somewhat protect against KCI attacks. Refer to section KCI attacks for an explanation . This long-lived key is serialized as follows:

```
OTRv4 public Ed448 forging key (ED448-FORGING-KEY):

  Pubkey type
    2 byte unsigned value, little-endian
    Ed448 public keys have type 0x0012

  F (POINT)
    F is the Ed448 public key generated as defined in RFC 8032, or directly
    as a random point.
```

The OTRv4 public shared prekey is defined as follows:

```
OTRv4 public shared prekey (ED448-SHARED-PREKEY):

  Shared Prekey type
```

```
  2 byte unsigned value, little-endian
  Ed448 shared prekeys have type 0x0011

D (POINT)
  D is the Ed448 shared prekey generated the same way as the public key in
  RFC 8032.
```

The public key and shared prekey are generated as follows (refer to RFC 8032 [9], for more information on key generation). Note that, although the RFC 8032 defines parameters as octet strings, they are defined as bytes here:

```
The symmetric key (sym_key) is 57 bytes of cryptographically secure random data.
If the symmetric key is used for the generation of 'ED448-PUBKEY', it is
denoted 'sym_h' If it is used for the generation of 'ED448-SHARED-PREKEY', it
is denoted 'sym_d'.

1. Hash the 'sym_key' using 'SHAKE-256(sym_key, 114)'. Store the digest in a
   114-byte buffer. Only the lower 57 bytes (denoted 'h') are used for
   generating the public key.
2. Prune the buffer 'h': the two least significant bits of the first
   byte are cleared, all eight bits of the last byte are cleared, and the
   highest bit of the second to last byte is set.
3. Interpret the buffer as the little-endian integer, forming the
   secret scalar 'sk'.  Perform a known-base-point scalar multiplication
   'sk * Base point (G)'. If the result is for the 'ED448-PUBKEY', store it in
   'H', encoded as POINT. If the result is for the 'ED448-SHARED-PREKEY',
   store it in 'D', encoded as POINT.
4. Securely store 'sk' locally, as 'sk_h' for 'ED448-PUBKEY', and 'sk_d' for
   'ED448-SHARED-PREKEY'. These keys will be stored for as long as the
   'ED448-PUBKEY' and the 'ED448-SHARED-PREKEY'  respectevely live.
   Additionally, securely store 'sym_key'. This key will be used for the Client
   and Prekey profiles signature. After their public key counterpart expires,
   they should be securely deleted or replaced.
5. Securely delete 'h'.
```

The forging keypair can be generated in one of two different ways - either by generating the key as detailed above (by using scalar multiplication), just like the 'honest' long lived Ed448 keys, or by directly generating a point on the curve (the long-term public key) without its corresponding secret, as detailed below. An implementation that uses the OTRv4 protocol must always implement the forging keys. It should ask users whether or not they would like to save the secret part of the forging keys (even if it was generated only as a point in the curve without a secret key counterpart). Even if most users select the default option to securely erase the forging keys, thereby preventing them from performing online forgery techniques, someone watching the protocol does not generally know the choice of the particular user. Consequently, someone that engages in a conversation using a compromised device is given two explanations: either the conversation is genuine, or the owner of the device is one of the users that elected to store the forgery keys and they are using those keys to forge the conversation. The result is that online deniability is preserved, while preventing KCI attacks.

In order to generate the forging key as a point directly on the curve:

either:

1. Generate 57 bytes of cryptographically secure random data (buf).
2. Deserialize buf into a POINT.
3. Check whether it is a valid point. See Verifying that a point is on the curve section for details.

or:

1. Use the Elligator technique [15] by mapping a string to valid Ed448 curve point.

Public keys (Ed448 public key) and forging keys (Ed448 public forging key) have fingerprints, which are hex strings that serve as identifiers. The full OTRv4 public key fingerprint is calculated by taking the SHAKE-256 hash of the byte-level representation of the public key and the byte-level representation of the forging public key. To authenticate the long-term key pairs, the Socialist Millionaire's Protocol or a manual fingerprint comparison may be used. The fingerprint is generated as:

- $\mathrm{HWC}(\mathrm{usage\_fingerprint} \,||\, \mathrm{byte}(H) \,||\, \mathrm{byte}(F), 56)$ (224-bit security level).

Instance Tags

Clients include instance tags in all OTRv4 messages. Instance tags are 4-byte values that are intended to be persistent. If the same client is logged into the same account from multiple locations/clients, the intention is that the client will have different instance tags at each location/client. OTRv4 messages (fragmented and unfragmented) include the source and destination instance tags. If a client receives a message that lists a destination instance tag different from its own, the client should discard the message.

The smallest valid instance tag is $0x00000100$. It is appropriate to set the destination instance tag to 0 when an actual destination instance

tag is not known at the time the message is prepared. If a client receives a message with the sender instance tag set to less than `0x00000100`, it should discard the message. Similarly, if a client receives a message with the recipient instance tag set to greater than `0` but less than `0x00000100`, it should discard the message.

This practice avoids an issue on IM networks that always relay all messages to all sessions of a client who is logged in multiple times. In this situation, OTR clients can attempt to establish an OTR session indefinitely if there are interleaving messages from each of the sessions.

TLV Record Types

Each TLV record is of the form:

```
Type (SHORT)
  The type of this record. Records with unrecognized types should be ignored
Length (SHORT)
  The length of the following field
Value (len BYTE) [where len is the value of the Length field]
  Any pertinent data for the record type
```

OTRv4 supports some TLV record types from OTRv3. The supported types are:

```
Type 0: Padding
  The value may be an arbitrary amount of data. This data should be ignored.
  This type can be used to disguise the length of a plaintext message.

Type 1: Disconnected
  If the participant requests to close the private connection, you may send a
  message (possibly with empty human-readable part) containing a record with
  this TLV type just before you discard the session keys, and transition to
  'START' state (see below). If you receive a TLV record of this type, you
  should transition to 'FINISHED' state (see below), and inform the participant
  that its correspondent has closed its end of the private connection, and the
  participant should do the same. Old mac keys can be attached to this TLV when
  the session is expired. This TLV should have the 'IGNORE_UNREADABLE' flag set.

Type 2: SMP Message 1
  The value represents the initial message of the Socialist Millionaires'
  Protocol (SMP). Note that this represents TLV type 2 and 7 from OTRv3.
  This TLV should have the 'IGNORE_UNREADABLE' flag set.

Type 3: SMP Message 2
  The value represents the second message in an instance of the SMP. This TLV
  should have the 'IGNORE_UNREADABLE' flag set.

Type 4: SMP Message 3
  The value represents the third message in an instance of the SMP. This TLV
  should have the 'IGNORE_UNREADABLE' flag set.

Type 5: SMP Message 4
  The value represents the final message in an instance of the SMP. This TLV
  should have the 'IGNORE_UNREADABLE' flag set.

Type 6: SMP Abort Message
  If the participant cancels the SMP prematurely or encounters an error in the
  protocol and cannot continue, you may send a message (possibly with an empty
  human-readable part) with this TLV type to instruct the other party's client
  to abort the protocol. The associated length should be zero and the
  associated value should be empty. If you receive a TLV of this type, you
  should change the SMP state to 'SMPSTATE_EXPECT1' (see below, in SMP section).
  This TLV should have the 'IGNORE_UNREADABLE' flag set.

Type 7: Extra symmetric key
  If you wish to use the extra symmetric key, compute it yourself as outlined
  in the section "Extra symmetric key". Then send this type 7 TLV to your peer
  to indicate that you'd like to use the extra symmetric key for something. The
  value of the TLV begins with a 4-byte indication of what this symmetric key
  will be used for (file transfer, voice encryption, etc). After that, the
  contents are use-specific (which file, etc): there are no predefined uses.
  Note that the value of the key itself is not placed into the TLV, your peer
  will compute it on its own. This TLV represents TLV type 8 from OTRv3.
  This TLV should have the 'IGNORE_UNREADABLE' flag set.
```

If you receive a data message with a corrupted TLV (an incomplete one or a TLV not included on this list), stop processing it, but process the remaining data message and any other TLVs included in it.

Shared Session State

Both the interactive and non-interactive DAKEs must authenticate their contexts to prevent attacks that rebind the DAKE transcript into different contexts. If the higher-level protocol ascribes some property to the connection, the DAKE exchange should verify this property, so both sides of a conversation can cryptographically verify some beliefs they have about the session.

A session is created when a new OTRv4 conversation begins. Given a shared session state information `phi` (e.g., a session identifier) associated with the higher-level context (e.g., XMPP), the DAKE authenticates that both parties share the same value for `phi` (Φ).

The shared session state (Φ) verifies shared state from the higher-level protocol as well as from OTR itself. Therefore, an implementer (who has complete knowledge of the application network stack) should define a known shared session state from the higher-level protocol as `phi'`, as well as include the values imposed by this specification.

Note that variable length fields are encoded as DATA. If `phi'` is a string, it will be encoded in UTF-8.

To make sure both participants has the same phi during DAKE, sort the instance tag by numerical order and any string passed to `phi'` lexicographically.

```
session identifier mandated by the OTRv4 spec = sender and receiver's instance
  tags, first ephemeral keys for the double ratchet initialization
phi' = session identifier defined by the implementer
phi = session identifier mandated by the OTRv4 spec || phi'
```

In XMPP, for example, `phi'` can be the node and domain parts of the sender and receiver's jabber identifier, e.g. `alice@jabber.net` (often referred as the "bare JID"). In an application that assigns some attribute to users before a conversation (e.g., a networked game in which players take on specific roles), the expected attributes (expressed in fixed length) should be included in `phi'`. A static password shared by both sides can also be included.

For example, a shared session state which higher-level protocol is XMPP, will look like:

```
phi = sender's instance tag || receiver's instance tag || our_ecdh_first ||
      our_dh_first || their_ecdh_first || their_dh_first ||
      DATA(sender's bare JID) || DATA(receiver's bare JID)
phi = 0x00000100 || 0x00000101 || 0x57 || 0x164 || 0x57 || 0x164 ||
      DATA("alice@jabber.net") || DATA("bob@jabber.net")
```

Secure Session ID

The secure session ID (`SSID`) is a 8-byte value. If the participant requests to see it, it should be displayed as two 4-byte big-endian unsigned values. For example, in C language, in "%08x" format. If the party transmitted the Auth-R message during the DAKE, then display the first 4 bytes in bold, and the second 4 bytes in non-bold. If the party transmitted the Auth-I message instead, display the first 4 bytes in non-bold, and the second 4 bytes in bold. If the party transmitted the Non-Interactive-Auth message during the DAKE, then display the first 4 bytes in bold, and the second 4 bytes in non-bold. If the party received the Non-Interactive-Auth message instead, display the first 4 bytes in non-bold, and the second 4 bytes in bold.

This Secure Session ID can be used by the parties to verify (over the telephone, assuming the parties recognize each others' voices) that there is no man-in-the-middle by having each side read his bold part to the other. Note that this only needs to be done in the event that the participants do not trust that their long-term keys have not been compromised.

OTR Error Messages

Any message containing "?OTR Error: " at the starting position is an OTR Error Message. The following part of the message should contain human-readable details of the error. The message may also include a specific code at the beginning, e.g. "?OTR Error: ERROR_N: ". This code is used to identify which error is being received for optional localization of the message.

Currently, the following errors are supported:

```
ERROR_1: (the message is undecryptable)
  Unreadable message
ERROR_2: (the message arrived in a state that is not encrypted messages)
  Not in private state message
ERROR_3: (the instance tags do not correspond)
  Malformed message
```

Note that the string "?OTR Error:" must be in at the start position of the message because of these reasons:

- The possibility for playing games with the state machine by "embedding" this string inside some other message.
- The potential of social engineering depending on the UI of the used chat client.

Key Management

In both the interactive and non-interactive DAKEs, OTRv4 uses long-term Ed448 keys, ephemeral Elliptic Curve Diffie-Hellman (ECDH) keys, and ephemeral Diffie-Hellman (DH) keys.

For exchanging data messages, OTRv4 uses KDF chains: the symmetric-key ratchet and the DH ratchet (with ECDH) from the Double Ratchet algorithm [2]. OTRv4 adds 3072-bit (384-byte) DH keys, called the brace key pair, to the Double Ratchet algorithm. These keys are used to protect transcripts of data messages in case ECC is broken. During the DAKE and initialization of the Double Ratchet Algorithm, both parties agree upon the first set of ECDH and DH keys. Then, during every third DH ratchet in the Double Ratchet, a new DH key is agreed upon. Between each DH brace key ratchet, both sides will conduct a symmetric brace key ratchet.

The following variables keep state as the ratchet moves forward:

```
State variables:
  i: the ratchet id.
  max_remote_i_seen: the maximum 'i' seen from the other participant.
  j: the sending message id.
  k: the receiving message id.
  pn: the number of messages in the previous DH ratchet.
  since_last_dh: a variable that keeps track of the last time a DH key was
                 generated.

Key variables:
  'root_key': the root key. If it is 'prev_root_key', it refers to the previous
              generated root key. If it is 'curr_root_key', it refers to the
              current root key.
  'chain_key_s[j]': the sending chain key for the sending message 'j'.
  'chain_key_r[k]': the receiving chain key for the receiving message 'k'.
  'our_ecdh': our current ECDH ephemeral key pair.
  'their_ecdh': their ECDH ephemeral public key.
  'our_dh': our DH ephemeral key pair.
  'their_dh': their DH ephemeral public key.
  'brace_key': either a hash of the shared DH key: 'KDF(usage_third_brace_key ||
   k_dh, 32)' (every third DH ratchet) or a hash of the previous 'brace_key:
   KDF(usage_brace_key || brace_key, 32)'
  'mac_keys_to_reveal': the MAC keys to be revealed in the first data message
    sent of the next ratchet.
  'skipped_MKenc': Dictionary of stored skipped-over message keys, indexed by
    'their_ecdh' and the message number ('j'). Raises and exception if too many
    elements are stored.
  'max_skip' a constant that specifies the maximum number of message keys
    that can be skipped in a ratchet. It should be set by the implementer. Take
    into account that it should be set high enough to tolerate routine lost or
    delayed messages, but low enough that a malicious sender can't trigger
    excessive recipient computation.
```

Depending on the event, the state variables are incremented and some key variable values are replaced:

- When you start a new Interactive DAKE by sending or receiving an Identity Message.
- When you complete the Interactive DAKE by sending an Auth-I Message.
- When you complete the Interactive DAKE by receiving and validating an Auth-I Message.
- When you start a new Non-interactive DAKE by publishing or retrieving a Prekey Ensemble.
- When you complete a Non-interactive DAKE by sending a Non-interactive-Auth Message.
- When you complete a Non-interactive DAKE by receiving and validating a Non-interactive-Auth Message.
- When you send a Data Message or receive a Data Message.
- When you send a TLV type 1 (Disconnected).

Generating ECDH and DH keys

```
generateECDH()
  - pick a random value r (57 bytes)
  - Hash the 'r' using 'SHAKE-256(r, 114)'. Store the digest in a
    114-byte buffer. Only the lower 57 bytes (denoted 'h') are used for
    generating the public key.
  - prune 'h': the two least significant bits of the first byte are cleared, all
    eight bits of the last byte are cleared, and the highest bit of the second
    to last byte is set.
  - Interpret the buffer as the little-endian integer, forming the secret scalar
    's'.
```

```
  - Securely delete 'r' and 'h'.
  - return our_ecdh.public = G * s, our_ecdh.secret = s

generateDH()
  - pick a random value r (80 bytes)
  - return our_dh.public = g3 ^ r, our_dh.secret = r
```

Shared Secrets

```
k_dh:
  The 3072-bit DH shared secret computed from a DH exchange, serialized as a
  big-endian unsigned integer.

brace_key:
  Either a hash of the shared DH key: 'KDF(usage_third_brace_key || k_dh, 32)'
  (every third DH ratchet) or a hash of the previous:
  'brace_key: KDF(usage_brace_key || brace_key, 32)'.

K_ecdh:
  The serialized ECDH shared secret computed from an ECDH exchange, serialized
  as a 'POINT'.

K:
  The Mixed shared secret is the final shared secret derived from both the
  brace key and ECDH shared secrets:
  'KDF(usage_shared_secret || K_ecdh || brace_key, 64)'.
```

Generating Shared Secrets

```
ECDH(a, B)
  K_ecdh = a * B
  if K_ecdh == 0 (check that it is an all-zero value)
     return error
  else
     return K_ecdh
```

Check, without leaking extra information about the value of `K_ecdh`, whether `K_ecdh` is the all-zero value and abort if so, as this process involves contributory behavior. Contributory behaviour means that both parties' private keys contribute to the resulting shared key. Since Ed448 have a cofactor of 4, an input point of small order will eliminate any contribution from the other party's private key. This situation can be detected by checking for the all-zero output.

```
DH(a, B)
  return k_dh = a ^ B
```

Rotating ECDH Keys and Brace Key as sender

Before sending the first reply (i.e. a new message considering a previous message has been received) or sending the first data message, the sender will rotate their ECDH keys and their brace key. This is for the computation of the Mixed shared secret `K` (see Deriving Double Ratchet Keys).

Before rotating the keys:

- Reset the sending message id (j) to 0.

To rotate the ECDH keys:

- Generate a new ECDH key pair and assign it to `our_ecdh = generateECDH()` (by securely replacing the old value).
- Calculate `K_ecdh = ECDH(our_ecdh.secret, their_ecdh)`.
- `i = i + 1`

To rotate the brace key:

- If `since_last_dh == 3`:
    - Generate the new DH key pair and assign it to `our_dh = generateDH()` (by securely replacing the old value).
    - Calculate `k_dh = DH(our_dh.secret, their_dh)`.
    - Calculate a `brace_key = KDF(usage_third_brace_key || k_dh, 32)`.
    - Securely delete `k_dh`.

    - Set since_last_dh to 0.

- Otherwise:

    - Derive and securely overwrite brace_key = KDF(usage_brace_key || brace_key, 32).
    - Increase since_last_dh by 1.

Rotating ECDH Keys and Brace Key as receiver

Every ratchet, the receiver will rotate their ECDH keys and their brace key. This is for the computation of the Mixed shared secret K (see Deriving Double Ratchet Keys).

Before rotating the keys:

- Reset the receiving message id (k) to 0.

To rotate the ECDH keys:

- Retrieve the ECDH key ('Public ECDH key') from the received data message and assign it to their_ecdh.
- Calculate K_ecdh = ECDH(our_ecdh.secret, their_ecdh).
- Securely delete our_ecdh.secret.

To rotate the brace key:

- If since_last_dh == 3:

    - Retrieve the DH key ('Public DH key') from the received data message and assign it to their_dh.
    - Calculate k_dh = DH(our_dh.secret, their_dh).
    - Calculate a brace_key = KDF(usage_third_brace_key || k_dh, 32).
    - Securely delete our_dh.secret and k_dh.

- Otherwise:

    - Derive and securely overwrite brace_key = KDF(usage_brace_key || brace_key, 32).
    - Increase since_last_dh by 1.

Deriving Double Ratchet Keys

To derive the next root key and the current chain key:

Note that if there is no previous root key (because this is the first ratchet), then keys are derived from the previous Mixed shared secret K (interpreted as prev_root_key) and the current Mixed shared secret K.

Depending if this is used to derive sending or receiving chain keys, the variable i should refer to j (for sending) and k (for receiving).

```
derive_ratchet_keys(purpose, prev_root_key, K):
  curr_root_key = KDF(usage_root_key || prev_root_key || K, 64)
  chain_key_purpose[i] = KDF(usage_chain_key || prev_root_key || K, 64)
  return curr_root_key, chain_key_purpose[i]
```

Calculating Encryption and MAC Keys

When sending or receiving data messages, you must calculate the message keys:

```
derive_enc_mac_keys(chain_key):
  MKenc = KDF(usage_message_key || chain_key, 64)
  MKmac = KDF(usage_MAC_key || MKenc, 64)
  return MKenc, MKmac
```

Resetting State Variables and Key Variables

The state variables are set to 0 and the key variables are set to NULL.

Session Expiration

OTRv4 can be vulnerable to a situation when an attacker capture some messages to compromise their ephemeral secrets at a later time.

To mitigate against this, message keys should be deleted regularly. OTRv4 implements this by detecting whether a new ECDH key has been generated within a certain amount of time. If it hasn't, the session is expired.

To expire a session:

1. Calculate the MAC keys corresponding to the stored message keys in the skipped_MKenc dictionary and put them on the old_mac_keys list (so they are revealed in TLV type 1 (Disconnected) message).
2. Send a Data message containing a TLV type 1 with empty payload - this is often referred to as 'Disconnected message' - with the old_mac_keys list attached to it.
3. Securely delete all keys and data associated with the conversation. This includes:
    1. The root key and all chain keys.
    2. All message keys and extra symmetric keys stored in the skipped_MKenc dictionary.
    3. The ECDH keys, DH keys and brace keys.
    4. The Secure Session ID (SSID) whose creation is described here and here, any old MAC keys that remain unrevealed, and the extra symmetric key if present.
    5. Reset the state and key variables, as defined in its section.
4. Transition the protocol state machine to FINISHED state.

The session expiration time is decided individually by each party so it is possible for one person to have an expiration time of two hours and the other party have it of two weeks. The client implementer should decide what the appropriate expiration time is for their particular circumstance.

The session expiration encourages keys to be deleted often at the cost of having lost messages whose MAC keys cannot be revealed. For example, if Alice sets her session expiration time to be 2 hours, in order to reset Alice's session expiration timer, Bob must create a reply and Alice must create a response to this reply. If this does not happen within two hours, Alice will expire her session and delete all keys associated with this conversation. If she receives a message from Bob after two hours, she will not be able to decrypt the message and thus she will not reveal the MAC key associated with it. Note, nevertheless, that the MAC keys corresponding to stored message keys (from messages that have not yet arrived) will be derived and revealed in the TLV type 1 that is sent.

It is also possible for the heartbeat messages to keep a session from expiring. Sticking with the above example of Alice's 2 hour session expiration time, Bob or Bob's client may send a heartbeat message every minute. In addition, Alice's client may send a heartbeat every five minutes. Thus, as long as both Bob and Alice's clients are online and sending heartbeat messages, Alice's session will not expire. But if Bob's client turns off or goes offline for at least two hours, Alice's session will expire.

The session expiration timer begins at different times for the sender and the receiver of the first data message in a conversation. The sender begins their timer as they send the first data message or as they attach an encrypted message to the Non-Interactive-Auth message. The receiver begins their timer when they receive this first data message.

Since the session expiration uses a timer, it can be compromised by clock errors. Some errors may cause the session to be deleted too early and result in undecryptable messages being received. Other errors may result in the clock not moving forward which would cause a session to never expire. To mitigate this, implementers should use secure and reliable clocks that can't be manipulated by an attacker.

Client Profile

OTRv4 introduces Client Profiles. A Client Profile has an arbitrary number of fields, but some fields are required. A Client Profile contains the Client Profile owner instance tag, an Ed448 long-term public key, the Ed448 long-term forging public key, information about supported versions, a profile expiration date, a signature of all these, and an optional transitional signature. Therefore, the Client Profile has variable length.

There are two instances of the Client Profile that should be generated. One is used for authentication in both DAKEs (interactive and non-interactive). The other should be published in a public place. This allows two parties to send and verify each other's Client Profiles during the DAKEs without damaging the deniability properties for the conversation, since the Client Profile is public information. A Client Profile is also published so it is easier to revoke any past value that could have been advertised on a previous Client Profile, to to prevent "version" rollback attacks, and to start offline conversations.

Each implementation should decide how to publish the Client Profile. For example, one client may publish profiles to a server pool (similar to a keyserver pool, where PGP public keys can be published). Another client may use XMPP's publish-subscribe extension (XEP-0060 [8]) for publishing Client Profiles. For sending offline messages, notice that the Client Profile has to be published and stored in the same untrusted Prekey Server used to store Prekey Messages and Prekey Profiles, so the Prekey Ensemble can be assembled.

A Client Profile has an expiration time as this helps to revoke any past value stated in a previous profile. If a user's client, for example, changes its long-term public key, only the valid non-expired Client Profile is the one used for attesting that this is indeed the valid long-term public key. Any expired Client Profiles with old long-term public keys are invalid. Moreover, as version advertisement is public information

(it is stated in the published Client Profile), a participant will not be able to delete this information from public servers (if the Client Profile is published in them). To facilitate version revocation or any of the other values revocation, the Client Profile can be regenerated and republished once the older Client Profile expires. This is also the reason why we recommend a short expiration date, so values can be easily revoked.

Before the Client Profile expires, it should be updated. Client implementations should determine the frequency of the Client Profile expiration and renewal. The recommended expiration time is one week.

Nevertheless, for a short amount of time (decided by the client) a Client Profile can still be locally used even if it has publicly expired. This is needed for non-interactive conversations as a party, Alice, can send offline encrypted messages using a non-expired published Client Profile from Bob. This Client Profile, nevertheless, can expire prior to the moment in which the other party, Bob, receives the offline encrypted messages. To allow this party, Bob, to still be able to validate and decrypt these messages, the Client Profile can still be locally used even if it has publicly expired. A recommended amount of time for this extra validity time is of 1 day.

It is also important to note that the absence of a Client Profile is not a proof that a user does not support OTRv4.

Notice that the valid lifetime of the long-term public key and forging public key is exactly the same as the lifetime of the Client Profile. If you have no valid Client Profile available for a specific long-term public key or for a specific forging public key, that long-term public key or forging public key should be treated as invalid. This also implies that a long term public key or forging public key can go from being valid to invalid, and back to valid. Notice, nevertheless, that long-term public keys and forging public keys can live longer than a Client Profile. A long-term public keys or forging public key does not need to be generated every time a Client Profile is renewed. But a long-term public key or a forging public key is only valid for the amount of time a Client Profile (that has them) is valid.

A Client Profile also includes an instance tag. This value is used for locally storing and retrieving the Client Profile during the non-interactive DAKE. This instance tag has to match the sender instance tag of the DAKE message the Client Profile is included in.

Note that a Client Profile is generated per client basis. Users are not expected to manage Client Profiles (theirs or from others) in a client. As a consequence, clients are discouraged to allow importing or exporting of Client Profiles. Also, if a user has multiple client locations concurrently in use, it is expected that they have multiple Client Profiles simultaneously published and valid.

A Client Profile can be used to prevent rollback attacks. As a query message can be intercepted and changed by a Man-in-the-Middle (MitM) to enforce the lowest version advertised, a participant can check for the published Client Profile to see if this is indeed the highest supported version.

Client Profile Data Type

```
Client Profile (CLIENT-PROF):
  Number of Fields (INT)
  Fields (SEQ-FIELDS)
    2 byte unsigned type, big-endian
    the encoded field
  Client Profile Signature (CLIENT-EDDSA-SIG)
```

The supported fields should not be duplicated. They are:

```
Client Profile owner instance tag (INT)
  Type = 0x0001
  The instance tag of the client/device that created the Client Profile.

Ed448 public key (ED448-PUBKEY)
  Type = 0x0002
  Corresponds to 'OTRv4 public authentication Ed448 key'.

Ed448 public forging key (ED448-FORGING-KEY)
  Type = 0x0003
  Corresponds to 'OTRv4 public Ed448 forging key'.

Versions (DATA)
  Type = 0x0004

Client Profile Expiration (CLIENT-PROF-EXP)
  Type = 0x0005

OTRv3 public authentication DSA key (PUBKEY)
  Type = 0x0006

Transitional Signature (CLIENT-SIG)
  Type = 0x0007
  This signature is defined as a signature over fields 0x0001,
  0x0002, 0x0003, 0x0004, 0x0005 and 0x0006 only.
```

Note that the Client Profile Expiration is encoded as:

```
Client Profile Expiration (CLIENT-PROF-EXP):
  8 bytes signed value, big-endian
```

CLIENT-EDDSA-SIG refers to the OTRv4 EDDSA signature:

```
EDDSA signature (CLIENT-EDDSA-SIG):
  (len is the expected length of the signature, which is 114 bytes)
  len byte unsigned value, little-endian
```

CLIENT-SIG is the DSA Signature. It is the same signature as used in OTRv3. From the OTRv3 protocol, section "Public keys, signatures, and fingerprints", the format for a signature generated with the OTRv3 DSA public key is as follows:

```
DSA signature (CLIENT-SIG):
  (len is the length of the DSA public parameter q, which in current
  implementations is 20 bytes)
  len byte unsigned r, big-endian
  len byte unsigned s, big-endian
```

If version 3 and 4 are supported, the OTRv3 DSA public key must be included in the Client Profile. As defined in OTRv3 spec, the OTRv3 DSA public key is defined as:

```
OTRv3 public authentication DSA key (PUBKEY):
  Pubkey type (SHORT)
    DSA public keys have type 0x0000
  p (MPI)
  q (MPI)
  g (MPI)
  y (MPI)
(p,q,g,y) are the OTRv3 DSA public key parameters
```

Creating a Client Profile

To create a Client Profile, generate:

1. A 4-byte instance tag to use as the Client Profile owner instance tag. This should only be done if the client doesn't already have an instance tag for this user.

Then, assemble:

1. Client Profile owner instance tag.
2. Ed448 long-term public key.
3. Ed448 long-term public forging key.
4. Versions: a string corresponding to the user's supported OTR versions. A Client Profile can advertise multiple OTR versions. The format is described under the section Establishing Versions below.
5. Client Profile Expiration: Expiration date in standard Unix 64-bit format (seconds since the midnight starting Jan 1, 1970, UTC, ignoring leap seconds).
6. OTRv3 public authentication DSA key (optional): The OTRv3 long-term public key. It should be included if version 3 and 4 are supported.
7. Transitional Signature (optional): A signature of the Client Profile excluding the Client Profile Signature and the user's OTRv3 DSA key. The Transitional Signature enables parties that trust user's version 3 DSA key to trust the Client Profile in version 4. This is only used if the user supports versions 3 and 4. For more information, refer to Create a Client Profile Signature section. The presence of OTRv3 public authentication DSA key means that the Transitional signature is mandatory. But the user can decide to have a Transitional signature without publish the OTRv3 public authentication DSA key on the Client Profile.

Then:

1. Assemble the previous fields as Fields.
2. Assign the number of Fields as $\mathrm{Number\ of\ Fields}$.
3. Generate the Client Profile signature: The symmetric key, the flag f (set to zero, as defined on RFC 8032 [9]) and the empty context c are used to create a signature of the entire Client Profile excluding the signature itself. The size of the signature is 114 bytes. For its generation, refer to Create a Client Profile Signature section.

After the Client Profile is created, it must be published in a public untrusted place. When using OTRv4 in OTRv3-compatible mode and OTRv4-standalone mode, notice that the Client Profile has to be published and stored in the untrusted Prekey Server used to store prekey

messages and Prekey Profiles.

Establishing Versions

A valid version string can be created by concatenating supported version numbers together in any order. For example, a user who supports versions 3 and 4 will have the 2-byte version string "43" or "34" in their Client Profile. A user who only supports version 4 will have the 1-byte version string "4". Thus, a version string has varying size, and it is represented as a DATA type with its length specified.

A compliant OTRv4 implementation (in OTRv4-compatible mode) is required to support version 3 of OTR, but not versions 1 and 2. Therefore, invalid version strings contain a "2" or a "1".

Any other version string that is not "4", "3", "2", or "1" should be ignored.

Client Profile Expiration and Renewal

If a renewed Client Profile is not published in a public place, the user's deniability is at risk. Deniability is also at risk if the only publicly available Client Profile is expired. For that reason, a received expired Client Profile during the DAKE meeting that criteria is considered invalid.

Before the Client Profile expires, the user must publish an updated Client Profile with a new expiration date. This expiration date is defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970.

The client establishes the frequency of expiration and when to publish (before the current Client Profile expires). Note that this can be configurable. A recommended value is one week.

Create a Client Profile Signature

If version 3 and 4 are supported and the user has a pre-existing OTRv3 long-term keypair:

- Concatenate $\text{Client Profile owner instance tag} \mid\mid \text{Ed448 public key} \mid\mid \text{Ed448 public forging key} \mid\mid \text{Versions} \mid\mid \text{Client Profile Expiration} \mid\mid \text{OTRv3 public authentication DSA key}$. Denote this value m.
- Sign m with the user's OTRv3 DSA key. Denote this value $\text{Transitional Signature}$.
- Sign $m \mid\mid \text{Transitional Signature}$ with the symmetric key, as stated below. Denote this value $\text{Client Profile Signature}$.

Note that if you only have version 4 turned on (on a local policy) but still support version 3 (you have a OTRv3 long-term keypair), you don't need to sign the Client Profile with this key.

If only version 4 is supported:

- Concatenate $\text{Client Profile owner's instance tag} \mid\mid \text{Ed448 public key} \mid\mid \text{Ed448 public forging key} \mid\mid \text{Versions} \mid\mid \text{Client Profile Expiration}$. Denote this value m.
- Sign m with the symmetric key, as stated below. Denote this value $\text{Client Profile Signature}$.

The Client Profile signature for version 4 is generated as defined in RFC 8032 [9], section 5.2.6. The flag f is set to 0 and the context c is an empty constant string.

Note that, although the RFC 8032 defines parameters as octet strings, they are defined as bytes here.

It is generated as follows:

```
The inputs are the symmetric key (57 bytes, defined in the 'Public keys and
fingerprints' section. It is referred as 'sym_key'), a flag 'f', which is a byte
with value 0, a context 'c' (a value set by the signer and verifier of maximum
255 bytes), which is an empty byte string for this protocol, and a message 'm'.
The function 'len(x)' should be interpreted here as the number of bytes in the
string 'x'.

1.  Hash the 'sym_key': 'SHAKE-256(sym_key, 114)'. Let 'h' denote the resulting
    digest. Construct the secret key 'sk' from the first half of 'h' (57 bytes),
    and the corresponding public key 'H', as defined in the 'Public keys, Shared
    Prekeys and Fingerprints' section. Let 'prefix' denote the second half of
    the 'h' (from 'h[57]' to 'h[113]').

2.  Compute 'SHAKE-256("SigEd448" || byte(f) || byte(len(c)) || c || prefix ||
    m, 114)', where 'm' is the message to be signed. Let 'r' be the 114-byte
    resulting digest and interpret it as a little-endian integer.

3.  Multiply the scalar 'r' by the Base Point (G). For efficiency, do this by
    first reducing 'r' modulo 'q', the group order.  Let 'R' be the encoding
    of this resulting point. It should be encoded as a POINT.
```

```
4.  Compute 'SHAKE-256("SigEd448" || f || len(c) || c || R || H || m, 114)'.
    Interpret the 114-byte digest as a little-endian integer 'k'.

5.  Compute 'S = (r + k * sk) mod q'.  For efficiency, reduce 'k' again modulo
    'q' first.

6.  Form the signature of the concatenation of 'R' (57 bytes) and the
    little-endian encoding of 'S' (57 bytes, the ten most significant bits are
    always zero).

7. Securely delete 'sk', 'h', 'r' and 'k'.
```

Verify a Client Profile Signature

The Client Profile signature is verified as defined in RFC 8032 [9], section 5.2.7. It works as follows:

```
1.  To verify a signature on a message 'm', using the public key 'H', with 'f'
    being 0, and 'c' being empty, split the signature into two 57-byte halves.
    Decode the first half as a point 'R', and the second half as a scalar
    'S'. Decode the public key 'H' as a point 'H_1'. If any of the
    decodings fail (including 'S' being out of range), the signature is invalid.

2.  Compute 'SHAKE-256("SigEd448" || byte(f) || byte(len(c)) || c || R || H ||
    m, 114)'. Interpret the 114-byte digest as a little-endian integer 'k'.

3.  Check the group equation '4 * (S * G) = (4 * R) + (4 * (k * H_1))'. It's is
    sufficient to check '(S * G) = R + (k * H_1)'.
```

Validating a Client Profile

To validate a Client Profile, you must (in this order):

1. Verify that the `Client Profile Signature` field is not empty.
2. Verify that the Client Profile signature is valid.
3. Verify that the Client Profile owner's instance tag is equal to the Sender Instance tag of the person that sent the DAKE message in which the Client Profile is received.
4. Verify that the Client Profile has not expired.
5. Verify that the `Versions` field contains the character "4".
6. Validate that `Ed448 Public Key` is on the curve Ed448-Goldilocks. See Verifying that a point is on the curve section for details.
7. Validate that `Ed448 Public Forging Key` is on the curve Ed448-Goldilocks. See Verifying that a point is on the curve section for details.
8. If the `Transitional Signature` and OTRv3 DSA key are present and, verify their validity using the OTRv3 DSA key. This validation is optional.

Prekey Profile

OTRv4 introduces prekey profiles. The Prekey Profile contains the Prekey Profile owner's instance tag, a shared prekey, a prekey profile expiration date and a signature of all these. It is signed by the Ed448 long-term public key.

A prekey profile is needed for it's signed shared prekey, which is used for offline conversations. It is changed on a regular basis as defined by the expiration date in it.

There are two instances of the Prekey Profile that should be generated. One is used for publication in an untrusted prekey server, so parties can use it to send offline messages. The other should be stored locally to be used in the Non-Interactive DAKE. Notice that the Prekey Profile has to be published and stored in the same untrusted Prekey Server used to store prekey messages. This is needed in order to generate the Prekey Ensemble needed for non-interactive conversations.

When the Prekey Profile expires, it should be updated. Client implementations should determine the frequency of the prekey's profile expiration and renewal. They should also determine when a new Prekey Profile is published (prior to it been expired or just at moment it expires). The recommended expiration time is one week.

Nevertheless, for a short amount of time (decided by the client) a Prekey Profile can still be locally used even if it has publicly expired. This is needed for non-interactive conversations as a party, Alice, can send offline encrypted messages using a non-expired published Prekey Profile from Bob. This Prekey Profile, nevertheless, can expire prior to the moment in which the other party, Bob, receives the offline encrypted messages. To allow this party, Bob, to still be able to validate and decrypt these messages, the Prekey Profile can still be locally used even if it has publicly expired. A recommended amount of time for this extra validity time is of 1 day.

Note that a Prekey Profile is generated per client location basis. Users are not expected to manage prekey profiles (theirs or from others)

in a client. As a consequence, clients are discouraged to allow importing or exporting of prekey profiles. Also, if a user has multiple client locations concurrently in use, it is expected that they have multiple prekey profiles simultaneously published and valid.

Prekey Profile Data Type

```
Prekey Profile Expiration (PREKEY-PROF-EXP):
  8 byte signed value, big-endian

Prekey Profile (PREKEY-PROF):
  Prekey Profile owner's instance tag (INT)
    The instance tag of the client that created the Prekey Profile.
  Prekey Profile Expiration (PREKEY-PROF-EXP)
  Public Shared Prekey (ED448-SHARED-PREKEY)
    The shared prekey used between different prekey messages.
    Corresponds to 'D'.
  Prekey Profile Signature (PREKEY-EDDSA-SIG)
    Created with the Ed448 long-term public key.
```

Note that the Prekey Profile Expiration is encoded as:

```
Client Profile Expiration (PREKEY-PROF-EXP):
  8 bytes signed value, big-endian
```

PREKEY-EDDSA-SIG refers to the OTRv4 EDDSA signature:

```
PREKEY-EDDSA-SIG signature (PREKEY-EDDSA-SIG):
  (len is the expected length of the signature, which is 114 bytes)
  len byte unsigned value, little-endian
```

Creating a Prekey Profile

To create a Prekey Profile, assemble:

1. The same Client Profile owner's instance tag. Denote this value Prekey Profile owner's instance tag.
2. Prekey Profile Expiration: Expiration date in standard Unix 64-bit format (seconds since the midnight starting Jan 1, 1970, UTC, ignoring leap seconds).
3. Public Shared Prekey: An Ed448 Public Key used in multiple prekey messages. It adds partial protection against an attacker that modifies the first flow of the non-interactive DAKE and that compromises the receivers long-term secret key and their one-time ephemeral keys. For its generation, refer to the Public keys, Shared Prekeys, Forging keys and Fingerprints section. This key must expire when the Prekey Profile expires.
4. Profile Signature: The symmetric key sym_h, the flag f (set to zero, as defined on RFC 8032 [9]) and the empty context c are used to create signatures of the entire profile excluding the signature itself. The size of the signature is 114 bytes. For its generation, refer to the Create a Prekey Profile Signature section.

After the Prekey Profile is created, it must be published in the untrusted Prekey Server.

Prekey Profile Expiration and Renewal

Before the prekey profile expires, the user must publish an updated prekey profile with a new expiration date. This expiration date is defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970.

The client establishes the frequency of expiration and when to publish (before the current Prekey Profile expires). Note that this can be configurable. A recommended value is one week.

Create a Prekey Profile Signature

For this:

- Concatenate $\text{Prekey Profile's owner's instance tag} \mid\mid \text{Prekey Profile Expiration} \mid\mid \text{Public Shared Prekey}$. Denote this value m.
- Sign m with the symmetric key sym_h, as stated below. Denote this value Profile Signature.

The Prekey Profile signature for version 4 is generated as defined in RFC 8032 [9], section 5.2.6. The flag f is set to 0 and the context c is an empty constant string.

Note that, although the RFC 8032 defines parameters as octet strings, they are defined as bytes here.

It is generated as follows:

```
The inputs are the symmetric key (57 bytes, defined in the 'Public keys and
fingerprints' section. It is referred as 'sym_key_h'), a flag 'f', which is a
byte with value 0, a context 'c' (a value set by the signer and verifier of
maximum 255 bytes), which is an empty byte string for this protocol, and a
message 'm'. The function 'len(x)' should be interpreted here as the number of
bytes in the string 'x'.

1.  Hash the 'sym_key_h': 'SHAKE-256(sym_key, 114)'. Let 'h' denote the
    resulting digest. Construct the secret key 'sk' from the first half of
    'h' (57 bytes), and the corresponding public key 'H', as defined in the
    'Public keys, Shared Prekeys, Forging keys and Fingerprints' section. Let
    'prefix' denote the second half of the 'h' (from 'h[57]' to 'h[113]').

2.  Compute 'SHAKE-256("SigEd448" || byte(f) || byte(len(c)) || c || prefix ||
    m, 114)', where 'm' is the message to be signed. Let 'r' be the 114-byte
    resulting digest and interpret it as a little-endian integer.

3.  Multiply the scalar 'r' by the Base Point (G). For efficiency, do this by
    first reducing 'r' modulo 'q', the group order.  Let 'R' be the encoding
    of this resulting point. It should be encoded as a POINT.

4.  Compute 'SHAKE-256("SigEd448" || f || len(c) || c || R || H || m, 114)'.
    Interpret the 114-byte digest as a little-endian integer 'k'.

5.  Compute 'S = (r + k * sk) mod q'.  For efficiency, reduce 'k' again modulo
    'q' first.

6.  Form the signature of the concatenation of 'R' (57 bytes) and the
    little-endian encoding of 'S' (57 bytes, the ten most significant bits are
    always zero).

7. Securely delete 'sk', 'h', 'r' and 'k'.
```

Verify a Prekey Profile Signature

The Prekey Profile signature is verified as defined in RFC 8032 [9], section 5.2.7. It works as follows:

```
1.  To verify a signature on a message 'm', using the long-term public key 'H',
    with 'f' being 0, and 'c' being empty, split the signature into two 57-byte
    halves. Decode the first half as a point 'R', and the second half as a
    scalar 'S'. Decode the public key 'H' as a point 'H_1'. If any of the
    decodings fail (including 'S' being out of range), the signature is invalid.

2.  Compute 'SHAKE-256("SigEd448" || byte(f) || byte(len(c)) || c || R || H ||
    m, 114)'. Interpret the 114-byte digest as a little-endian integer 'k'.

3.  Check the group equation '4 * (S * G) = (4 * R) + (4 * (k * H_1))'. It's is
    sufficient to check '(S * G) = R + (k * H_1)'.
```

Validating a Prekey Profile

To validate a Prekey Profile, you must (in this order):

1. Verify that the Prekey Profile signature is valid.
2. Verify that the Prekey Profile owner's instance tag is equal to the Sender Instance tag of the person that sent the DAKE message in which the Prekey Profile is received.
3. Verify that the Prekey Profile has not expired.
4. Verify that the Prekey Profile owner's instance tag is equal to the Sender Instance tag of the person that sent the DAKE message in which the Client Profile is received.
5. Validate that the `Public Shared Prekey` is on the curve Ed448-Goldilocks. See Verifying that a point is on the curve section for details.

Online Conversation Initialization

Online OTRv4 conversations are initialized through a Query Message or a Whitespace Tag. After this, the conversation is authenticated using the interactive DAKE.

Requesting Conversation with Older OTR Versions

Bob might respond to Alice's request (or notification of willingness to start a conversation) using OTRv3. If this is the case and Alice supports version 3, the protocol falls back to OTRv3 [7]. If Alice does not support version 3, this response is ignored.

Interactive Deniable Authenticated Key Exchange (DAKE)

This section outlines the flow of the interactive DAKE. This is a way to mutually agree upon shared keys for the two parties and authenticate one another while providing participation deniability.

This protocol is derived from the DAKEZ protocol [1], which uses a ring signature non-interactive zero-knowledge proof of knowledge (RING-SIG) for authentication (RSig).

Alice's long-term Ed448 key pair is (sk_ha, Ha) and Bob's long-term Ed448 keypair is (sk_hb, Hb). Both keypairs are generated as stated in the Public keys, shared prekeys and Fingerprints section. Alice and Bob also have long-term Ed448 forging public keys. These are denoted Fa for Alice's, and Fb for Bob's.

Interactive DAKE Overview

```
Alice                                   Bob
-----------------------------------------------------
      Query Message or Whitespace Tag -------->
      <---------------------- Identity message
      Auth-R ------------------------------->
      <------------------------------- Auth-I
```

Bob will be initiating the DAKE with Alice.

Bob:

1. Generates an Identity message, as defined in Identity Message section.
2. Sets Y and y as our_ecdh: the ephemeral ECDH keys.
3. Sets B as and b as our_dh: the ephemeral 3072-bit DH keys.
4. Sends Alice the Identity message with his our_ecdh_first.public and our_dh_first.public attached.

Alice:

1. Receives an Identity message from Bob:
   - Verifies the Identity message as defined in the Identity message section. If the verification fails (for example, if Bob's public keys -Y or B- are not valid), rejects the message and does not send anything further.
   - Picks the newest compatible version of OTR listed in Bob's profile. If there aren't any compatible versions, Alice does not send any further messages.
   - Sets Y as their_ecdh.
   - Sets B as their_dh.
   - Sets the received our_ecdh_first.public from Bob as their_ecdh_first.
   - Sets the received our_dh_first.public from Bob as their_dh_first.
2. Generates an Auth-R message, as defined in the Auth-R Message section.
3. Sets X and x as our_ecdh: the ephemeral ECDH keys.
4. Sets A and a as our_dh: the ephemeral 3072-bit DH keys.
5. Calculates the Mixed shared secret (K) and the SSID:
   - Calculates ECDH shared secret K_ecdh = ECDH(our_ecdh.secret, their_ecdh). Securely deletes our_ecdh.secret, our_ecdh.public and their_ecdh. Replaces them with:
     - our_ecdh.secret = our_ecdh_first.secret.
     - our_ecdh.public = our_ecdh_first.public.
     - their_ecdh = their_ecdh_first.
     - Securely deletes our_ecdh_first.secret, our_ecdh_first.public and their_ecdh_first.
   - Calculates DH shared secret k_dh = DH(our_dh.secret, their_dh). Securely deletes our_dh.secret, our_dh.public and their_dh. Replaces them with:
     - our_dh.secret = our_dh_first.secret.
     - our_dh.public = our_dh_first.public.
     - their_dh = their_dh_first.
     - Securely deletes our_dh_first.secret, our_dh_first.public and their_dh_first.
   - Calculates the brace key brace_key = KDF(usage_third_brace_key || k_dh, 32). Securely deletes k_dh.
   - Calculates the Mixed shared secret K = KDF(usage_shared_secret ||K_ecdh || brace_key, 64). Securely deletes K_ecdh and brace_key.
   - Calculates the SSID from shared secret: HWC(usage_SSID || K, 8).
6. Sends Bob the Auth-R message (see Auth-R Message section), with her our_ecdh_first.public and our_dh_first.public attached. Alice must not send data messages at this point, as she still needs to receive the 'Auth-I' message from Bob.

Bob:

1. Receives the Auth-R message from Alice:
    - Picks a compatible version of OTR listed on Alice's profile, and follows the specification for this version. If the versions are incompatible, Bob does not send any further messages.
2. Verifies the Auth-R message as defined in the Auth-R Message section. If the verification fails (for example, if Alice's public keys -X or A- are not valid), rejects the message and does not send anything further.
    - Sets X as their_ecdh.
    - Sets A as their_dh.
    - Sets the received our_ecdh_first.public from Alice as their_ecdh_first.
    - Sets the received our_dh_first.public from Alice as their_dh_first.
3. Creates an Auth-I message (see Auth-I Message section).
4. Calculates the Mixed shared secret (K) and the SSID:
    - Calculates ECDH shared secret K_ecdh = ECDH(our_ecdh.secret, their_ecdh). Securely deletes our_ecdh.secret, our_ecdh.public and their_ecdh. Replaces them with:
        - our_ecdh.secret = our_ecdh_first.secret.
        - our_ecdh.public = our_ecdh_first.public.
        - their_ecdh = their_ecdh_first.
        - Securely deletes our_ecdh_first.secret, our_ecdh_first.public and their_ecdh_first.
    - Calculates DH shared secret k_dh = DH(our_dh.secret, their_dh). Securely deletes our_dh.secret, our_dh.public and their_dh. Replaces them with:
        - our_dh.secret = our_dh_first.secret.
        - our_dh.public = our_dh_first.public.
        - their_dh = their_dh_first.
        - Securely deletes our_dh_first.secret, our_dh_first.public and their_dh_first.
    - Calculates the brace key brace_key = KDF(usage_third_brace_key || k_dh, 32). Securely deletes k_dh.
    - Calculates the Mixed shared secret K = KDF(usage_shared_secret || K_ecdh || brace_key, 64). Securely deletes k_ecdh and brace_key.
    - Calculates the SSID from shared secret: HWC(usage_SSID || K, 8).
5. Initializes the double-ratchet algorithm:
    - Sets since_last_dh as 0.
    - Sets i, j, k pn as 0.
    - Sets max_remote_i_seen as -1.
    - Interprets K as the first root key (prev_root_key) by: prev_root_key = KDF(usage_first_root_key || K, 64).
    - Calculates the receiving keys:
        - Calculates K_ecdh = ECDH(our_ecdh.secret, their_ecdh).
        - Calculates k_dh = DH(our_dh.secret, their_dh).
        - Calculates brace_key = KDF(usage_third_brace_key || k_dh, 32).
        - Securely deletes k_dh.
        - Calculates the Mixed shared secret (and replaces the old value) K = KDF(usage_shared_secret || K_ecdh || brace_key, 64). Securely deletes K_ecdh.
        - Derives new set of keys: curr_root_key, chain_key_r[k] = derive_ratchet_keys(receiving, prev_root_key, K).
        - Securely deletes the previous root key (prev_root_key) and K.
    - Calculates the sending keys:
        - Generates a new ECDH key pair and assigns it to our_ecdh = generateECDH() (by securely replacing the old value).
        - Calculates K_ecdh = ECDH(our_ecdh.secret, their_ecdh).
        - Generates the new DH key pair and assigns it to our_dh = generateDH() (by securely replacing the old value).
        - Calculates k_dh = DH(our_dh.secret, their_dh).
        - Calculates brace_key = KDF(usage_third_brace_key || k_dh, 32).
        - Securely deletes k_dh.
        - Calculates the Mixed shared secret (and replaces the old value) K = KDF(usage_shared_secret || K_ecdh || brace_key, 64). Securely deletes K_ecdh.
        - Interprets curr_root_key as prev_root_key.
        - Derives new set of keys: curr_root_key, chain_key_s[j] = derive_ratchet_keys(sending, prev_root_key, K).
        - Securely deletes the previous root key (prev_root_key) and K.
        - Increments since_last_dh = since_last_dh + 1.
        - Increments i = i + 1.
6. Sends Alice the Auth-I message (see Auth-I message section).
7. At this point, the interactive DAKE is complete for Bob:
    - In the case that he wants to immediately send a data message:

        - Follows what is defined in the When you send a Data Message section. Note that he will not perform a new DH ratchet, but rather start using the derived chain_key_s[j]. He should follow the "When sending a data message in the same DH Ratchet:" subsection and attaches his ECDH and DH public keys to this message.
    - In the case that he receives a data message:
        - Follows what is defined in the When you receive a Data Message section. Note that he will use the already derived chain_key_r[k]. He should follow the "Try to decrypt the message with a stored skipped message key" or "When receiving a data message in the same DH Ratchet:" subsections.

Alice:

1. Receives the Auth-I message from Bob:
    - Verifies the Auth-I message as defined in the Auth-I message section. If the verification fails, rejects the message and does not send anything further.
2. Initializes the double-ratchet algorithm:
    - Sets since_last_dh as 0.
    - Sets i, j, k and pn as 0.
    - Sets max_remote_i_seen as -1.
    - Interprets K as the first root key (prev_root_key) by: KDF(usage_first_root_key || K, 64).
    - Calculates the sending keys:
        - Calculates K_ecdh = ECDH(our_ecdh.secret, their_ecdh).
        - Calculates k_dh = DH(our_dh.secret, their_dh).
        - Calculates brace_key = KDF(usage_third_brace_key || k_dh, 32).
        - Securely deletes k_dh.
        - Calculates the Mixed shared secret (and replaces the old value): K = KDF(usage_shared_secret || K_ecdh || brace_key, 64). Securely deletes K_ecdh.
        - Derives new set of keys: curr_root_key, chain_key_s[j] = derive_ratchet_keys(sending, prev_root_key, K).
        - Securely deletes the previous root key (prev_root_key) and K.
        - Increments i = i + 1.
3. At this point, the interactive DAKE is complete for Alice:
    - In the case that she wants to immediately send a data message:
        - Follows what is defined in the When you send a Data Message section. Note that she will not perform a new DH ratchet, but rather use the already derived chain_key_s[j]. She should follow the "When sending a data message in the same DH Ratchet:" subsection.
    - In the case that she immediately receives a data message:
        - Follows what is defined in the When you receive a Data Message section. Note that she will perform a new DH ratchet with the advertised keys from Bob attached in the message. If she wants to send data messages at this point (after receiving ones), she will perform a new DH ratchet as well.

Identity Message

This is the first message of the DAKE. It is sent to commit to a choice of ECDH and DH key.

A valid Identity message is generated as follows:

1. If there is not a valid Client Profile, create a Client Profile, as defined in Creating a Client Profile section. Otherwise, use the one you have on local storage.
2. Generate an ephemeral ECDH key pair, as defined in Generating ECDH and DH keys:
    - secret key y (57 bytes).
    - public key Y.
3. Generate an ephemeral DH key pair, as defined in Generating ECDH and DH keys:
    - secret key b (80 bytes).
    - public key B.
4. Generate another ephemeral ECDH key pair, as defined in Generating ECDH and DH keys:
    - secret key our_ecdh_first.secret (57 bytes).
    - public key our_ecdh_first.public.
5. Generate another ephemeral DH key pair, as defined in Generating ECDH and DH keys:
    - secret key our_dh_first.secret (80 bytes).
    - public key our_dh_first.public.
6. Generate a 4-byte instance tag to use as the sender's instance tag. Additional messages in this conversation will continue to use this tag as the sender's instance tag. Also, this tag is used to filter future received messages. Messages intended for this instance of the client will have this number as the receiver's instance tag.

To verify an Identity message:

When receiving an Identity message, note that the participant must not start sending data messages, as they still need to receive the 'Auth-I' message.

1. Verify if the message type is 0x35.
2. Verify that protocol's version of the message is 0x0004.
3. Validate the Client Profile, as defined in Validating a Client Profile section.
4. Verify that the point Y received is on curve Ed448. See Verifying that a point is on the curve section for details.
5. Verify that the DH public key B is from the correct group. See Verifying that an integer is in the DH group section for details.
6. Verify that the point our_ecdh_first.public received is on curve Ed448. See Verifying that a point is on the curve section for details.
7. Verify that the DH public key our_dh_first.public is from the correct group. See Verifying that an integer is in the DH group section for details.

An Identity message is an OTRv4 message encoded as:

```
Protocol version (SHORT)
  The version number of this protocol is 0x0004.

Message type (BYTE)
  The message has type 0x35.

Sender's instance tag (INT)
  The instance tag of the person sending this message.

Receiver's instance tag (INT)
  The instance tag of the intended recipient. As the instance tag is used to
  differentiate the clients that a participant uses, this will often be 0 since
  the other party may not have set its instance tag yet.

Sender's Client Profile (CLIENT-PROF)
  As described in the section "Creating a Client Profile".

Y (POINT)
  The ephemeral public ECDH key.

B (MPI)
  The ephemeral public DH key. Note that even though this is in uppercase, this
  is NOT a POINT.

our_ecdh_first.public
  The ephemeral public ECDH key that will be used for the intialization of
  the double ratchet algorithm.

our_dh_first.public
  The ephemeral public DH key that will be used for the intialization of
  the double ratchet algorithm.
```

Auth-R Message

This is the second message of the DAKEZ. It is sent to commit to a choice of a ECDH ephemeral key and a DH ephemeral key, and to acknowledge the other participant's ECDH ephemeral key and DH ephemeral key. This acknowledgment includes a validation that other participant's ECDH key is on curve Ed448 and that its DH key is in the correct group.

A valid Auth-R message is generated as follows:

1. If there is not a valid Client Profile, create a Client Profile, as detailed as defined in Creating a Client Profile section. Otherwise, use the one you have on local storage.
2. Generate an ephemeral ECDH key pair, as defined in Generating ECDH and DH keys:
   - secret key x (57 bytes).
   - public key X.
3. Generate an ephemeral DH key pair, as defined in Generating ECDH and DH keys:
   - secret key a (80 bytes).
   - public key A.
4. Generate another ephemeral ECDH key pair, as defined in Generating ECDH and DH keys:
   - secret key our_ecdh_first.secret (57 bytes).
   - public key our_ecdh_first.public.
5. Generate another ephemeral DH key pair, as defined in Generating ECDH and DH keys:
   - secret key our_dh_first.secret (80 bytes).
   - public key our_dh_first.public.
6. Compute t = 0x0 || HWC(usage_auth_r_bob_client_profile || Bob_Client_Profile, 64) || HWC(usage_auth_r_alice_client_profile || Alice_Client_Profile, 64) || Y || X || B || A || HWC(usage_auth_r_phi || phi, 64). phi is the shared session state as mention in its section.

7. Compute sigma = RSig(H_a, sk_ha, {F_b, H_a, Y}, t), as defined in Ring Signature Authentication.
8. Generate a 4-byte instance tag to use as the sender's instance tag. Additional messages in this conversation will continue to use this tag as the sender's instance tag. Also, this tag is used to filter future received messages. For the other party, this will be the receiver's instance tag.
9. Use the sender's instance tag from the Identity Message as the receiver's instance tag.

To verify an Auth-R message:

1. Verify if the message type is 0x36.
2. Verify that protocol's version of the message is 0x0004.
3. Check that the receiver's instance tag matches your sender's instance tag.
4. Validate the Client Profile as defined in Validating a Client Profile section. Extract Ha from it.
5. Verify that the point X received is on curve Ed448. See Verifying that a point is on the curve section for details.
6. Verify that the DH public key A is from the correct group. See Verifying that an integer is in the DH group section for details.
7. Verify that the point our_ecdh_first.public received is on curve Ed448. See Verifying that a point is on the curve section for details.
8. Verify that the DH public key our_dh_first.public is from the correct group. See Verifying that an integer is in the DH group section for details.
9. Compute t = 0x0 || HWC(usage_auth_r_bob_client_profile || Bob_Client_Profile, 64) || HWC(usage_auth_r_alice_client_profile || Alice_Client_Profile, 64) || Y || X || B || A || HWC(usage_auth_r_phi || phi, 64). phi is the shared session state as mention in its section.
10. Verify the sigma as defined in Ring Signature Authentication: RVrf({F_b, H_a, Y}, sigma, t).

An Auth-R message is an OTRv4 message encoded as:

```
Protocol version (SHORT)
  The version number of this protocol is 0x0004.

Message type (BYTE)
  The message has type 0x36.

Sender's instance tag (INT)
  The instance tag of the person sending this message.

Receiver's instance tag (INT)
  The instance tag of the intended recipient.

Sender's Client Profile (CLIENT-PROF)
  As described in the section "Creating a Client Profile".

X (POINT)
  The ephemeral public ECDH key.

A (MPI)
  The ephemeral public DH key. Note that even though this is in uppercase, this
  is NOT a POINT.

sigma (RING-SIG)
  The 'RING-SIG' proof of authentication value.

our_ecdh_first.public
  The ephemeral public ECDH key that will be used for the intialization of
  the double ratchet algorithm.

our_dh_first.public
  The ephemeral public DH key that will be used for the intialization of
  the double ratchet algorithm.
```

Auth-I Message

This is the final message of the DAKE. It is sent to verify the authentication sigma.

A valid Auth-I message is generated as follows:

1. Compute t = 0x1 || HWC(usage_auth_i_bob_client_profile || Bobs_Client_Profile, 64) || HWC(usage_auth_i_alice_client_profile || Alice_Client_Profile, 64) || Y || X || B || A || HWC(usage_auth_i_phi || phi, 64). phi is the shared session state as mention in its section.
2. Compute sigma = RSig(H_b, sk_hb, {H_b, F_a, X}, t), as defined in Ring Signature Authentication.
3. Continue to use the sender's instance tag.

To verify an Auth-I message:

1. Verify if the message type is 0x37.
2. Verify that protocol's version of the message is 0x0004.
3. Check that the receiver's instance tag matches your sender's instance tag.

4. Compute t = 0x1 || HWC(usage_auth_i_bob_client_profile || Bobs_Client_Profile, 64) || HWC(usage_auth_i_alice_client_profile || Alices_Client_Profile, 64) || Y || X || B || A || HWC(usage_auth_i_phi || phi, 64). phi is the shared session state as mention in its section.
5. Verify the sigma as defined in Ring Signature Authentication: RVrf({H_b, F_a, X}, sigma, t).

An Auth-I is an OTRv4 message encoded as:

```
Protocol version (SHORT)
  The version number of this protocol is 0x0004.

Message type (BYTE)
  The message has type 0x37.

Sender's instance tag (INT)
  The instance tag of the person sending this message.

Receiver's instance tag (INT)
  The instance tag of the intended recipient.

sigma (RING-SIG)
  The 'RING-SIG' proof of authentication value.
```

Offline Conversation Initialization

To begin an offline conversation, a set of prekey messages, a Client Profile and a Prekey Profile are published to an untrusted Prekey Server. These three publications are defined as a Prekey Ensemble. This action is considered as the start of the non-interactive DAKE. A Prekey Ensemble is retrieved by the party attempting to send a message to the Prekey Ensemble publisher. This participant, then, replies with a Non-Interactive-Auth message (created with the Prekey Ensemble values). This action is considered to complete the non-interactive DAKE.

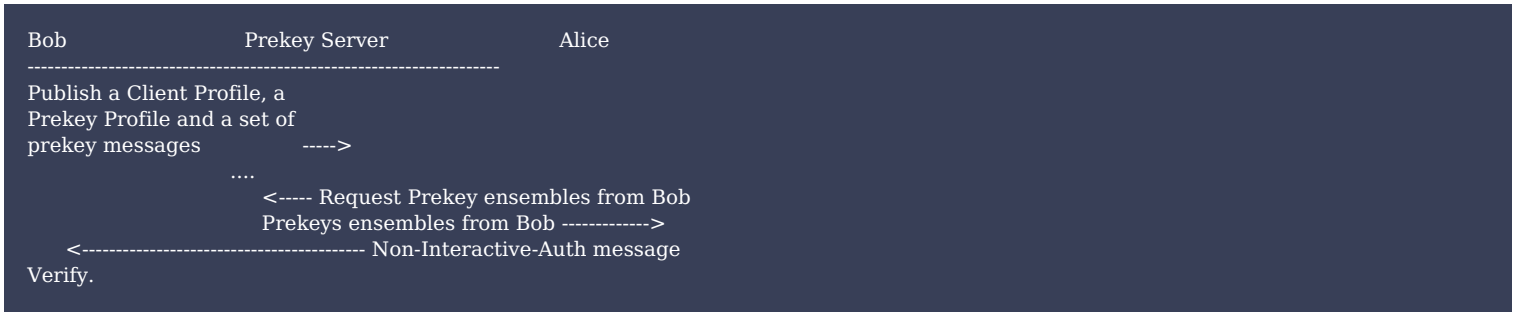
Non-interactive Deniable Authenticated Key Exchange (DAKE)

The non-interactive DAKE is a method by which two parties mutually agree upon shared cryptographic keys while providing partial participation deniability. Unlike the interactive DAKE, the non-interactive DAKE does not provide online deniability for the party that completes the DAKE by sending a Non-Interactive-Auth message. Client implementations are expected to understand this deniability risk when allowing participants to complete a non-interactive DAKE. They are also expected to decide how to convey this security loss to the participant.

This protocol is derived from the XZDH protocol [1], which uses a ring signature non-interactive zero-knowledge proof of knowledge (RING-SIG) for authentication (RSig).

Alice's long-term Ed448 key pair is (sk_ha, Ha) and Bob's long-term Ed448 key pair is (sk_hb, Hb). Both key pairs are generated as stated in the Public keys, Shared prekeys and Fingerprints section. Alice and Bob also have long-term Ed448 forging public keys. These are denoted Fa for Alice's, and Fb for Bob's.

Non-Interactive DAKE Overview




```
Bob                     Prekey Server                     Alice
-------------------------------------------------------------------
Publish a Client Profile, a
Prekey Profile and a set of
prekey messages               ----->
                          ....
                              <----- Request Prekey ensembles from Bob
                              Prekeys ensembles from Bob ------------>
    <------------------------------------------ Non-Interactive-Auth message
Verify.
```

Bob:

1. If there is not a valid Client Profile, creates a Client Profile, as defined in Creating a Client Profile section. Otherwise, use the one you have on local storage.
2. If there is not a valid Prekey Profile, creates a Prekey Profile, as defined in Creating a Prekey Profile section. Otherwise, use the one you have on local storage.
3. Generates prekey messages, as defined in the Prekey Message section.
4. Publishes the Client Profile, the Prekey Profile and the prekey messages to an untrusted Prekey Server. Note that he needs to publish a Client and Prekey Profile once for every long-term public key he locally has until the profiles respectively expire. He may upload new prekey messages at other times. See Publishing Prekey Ensembles section for details.

Alice:

1. Requests prekey ensembles from the untrusted server.
2. For each Prekey Ensemble received from the server:
    - Validates each Prekey Ensemble. If the verification fails, rejects the message and does not send anything further.
    - Picks a compatible version of OTR listed in Bob's Client Profile. If the versions are incompatible, Alice does not send any further messages.
    - Sets the received ECDH ephemeral public key Y as $their\_ecdh$.
    - Sets the received DH ephemeral public key B as $their\_dh$.
3. Extracts the Public Shared Prekey (D_b) from Bob's Prekey Profile. Extracts the Ed448 public key (H_b) from Bob's Client Profile. Sets the first as $their\_shared\_prekey$.
4. Generates a Non-Interactive-Auth message. See Non-Interactive-Auth Message section.
5. Sets X and x as our_ecdh: the ephemeral ECDH keys.
6. Sets A and a as our_dh: ephemeral 3072-bit DH keys.
7. Calculates the Mixed shared secret (K) and the SSID:
    - Gets tmp_k generated during the generation of the Non-Interactive-Auth Message.
    - Calculates the Mixed shared secret $K = KDF(usage\_shared\_secret \mid\mid tmp\_k, 64)$. Securely deletes tmp_k and $brace\_key$.
    - Calculates the SSID from shared secret: $HWC(usage\_SSID \mid\mid K, 8)$.
    - Securely deletes $our\_ecdh.secret$, $our\_ecdh.public$. Replaces them with:
        - $our\_ecdh.secret = our\_ecdh\_first.secret$.
        - $our\_ecdh.public = our\_ecdh\_first.public$.
        - Securely deletes $our\_ecdh\_first.secret$ and $our\_ecdh\_first.public$.
    - Securely deletes $our\_dh.secret$ and $our\_dh.public$. Replaces them with:
        - $our\_dh.secret = our\_dh\_first.secret$.
        - $our\_dh.public = our\_dh\_first.public$.
        - Securely deletes $our\_dh\_first.secret$ and $our\_dh\_first.public$.
8. Calculates the $Auth\ MAC$:
    - Calculates the value: $Auth\ MAC = HCMAC(usage\_auth\_MAC \mid\mid auth\_mac\_k \mid\mid t, 64)$
    - Includes this value in the Non-Interactive-Auth message and securely deletes the $auth\_mac\_k$.
9. Sends Bob a Non-Interactive-Auth message with her $our\_ecdh.public$ and $our\_dh.public$ attached. See Non-Interactive-Auth Message section.
10. Initializes the double-ratchet:
    - Sets $since\_last\_dh$ as 0.
    - Sets i, j, k and pn as 0.
    - Sets $max\_remote\_i\_seen$ as -1.
    - Calculates the root key and sending chain key: $curr\_root\_key = KDF(usage\_root\_key \mid\mid K, 64)$ and $chain\_key\_sending[j] = KDF(usage\_chain\_key \mid\mid K, 64)$.
    - Increments $i = i + 1$.
11. At this point, the non-interactive DAKE is complete for Alice:
    - If she wants to send a data message, she follows what is defined in the When you send a Data Message section. Note that she will not perform a new DH ratchet for this message, but rather use the already derived $chain\_key\_sending[j]$. She should follow the "When sending a data message in the same DH Ratchet:" subsection.

Bob:

1. Receives the Non-Interactive-Auth message from Alice:
    - Check that the receiver's instance tag matches your prekey message sender's instance tag.
    - Verify if the message type is $0x0D$.
    - Verify that protocol's version of the message is $0x0004$.
    - Validate the received ECDH ephemeral public key X is on curve Ed448. See Verifying that a point is on the curve section for details.
    - Validate that the received DH ephemeral public key A is on the correct group. See Verifying that an integer is in the DH group section for details.
    - Verify that the point $our\_ecdh\_first.public$ received is on curve Ed448. See Verifying that a point is on the curve section for details.
    - Verify that the DH public key $our\_dh\_first.public$ is from the correct group. See Verifying that an integer is in the DH group section for details.
    - Validates Alice's Client Profile and extracts H_a and F_a from it.
    - Retrieves his corresponding Prekey message from local storage, by using the 'Prekey Indentifier' attached to the Non-Interactive-Auth message.
        - If this 'Prekey Identifier' does not correspond to any Prekey message on local storage:
        - Aborts the DAKE.

        - Otherwise:
        - Sets Y and y as our_ecdh: the ephemeral ECDH keys.
        - Sets B as and b as our_dh: the ephemeral 3072-bit DH keys.
    - Retrieves his corresponding Client Profile from local storage.
        - If there is no Client Profile in local storage:
        - Aborts the DAKE.
        - Sets the 'Ed448 Long-term Public Key' from the Client Profile as Hb.
    - Retrieves his corresponding Prekey Profile from local storage.
        - If there is no Prekey Profile in local storage:
        - Aborts the DAKE.
        - Sets his Public Shared Prekey (D_b) from his Prekey Profile as our_shared_prekey.public.
    - Picks a compatible version of OTR listed on Alice's Client Profile, and follows the specification for this version. If the versions are incompatible, Bob does not send any further messages.
    - Verifies the Non-Interactive-Auth message. See Non-Interactive-Auth Message section. If the verification fails, rejects the message and does not send anything further.
2. Retrieves the ephemeral public keys from Alice:
    - Sets the received ECDH ephemeral public key X as their_ecdh.
    - Sets the received DH ephemeral public key A as their_dh.
    - Sets the received our_ecdh_first.public from Alice as their_ecdh_first.
    - Sets the received our_dh_first.public from Alice as their_dh_first.
3. Calculates the keys needed for the generation of the Mixed shared secret (K):
    - Calculates the ECDH shared secret K_ecdh = ECDH(our_ecdh.secret, their_ecdh). Securely deletes our_ecdh.secret.
    - Calculates the DH shared secret k_dh = DH(our_dh.secret, their_dh). Securely deletes our_dh.secret.
    - Calculates the brace key brace_key = KDF(usage_third_brace_key || k_dh, 32). Securely deletes k_dh.
4. Calculates tmp_k = KDF(usage_tmp_key || K_ecdh || ECDH(our_shared_prekey.secret, their_ecdh) || ECDH(sk_hb, their_ecdh) || brace_key, 64). Securely deletes K_ecdh.
5. Computes the Auth MAC key auth_mac_k = KDF(usage_auth_MAC_key || tmp_k, 64):
    - Computes Auth MAC = HCMAC(usage_auth_MAC || auth_mac_k || t, 64). The t value here is the one computed during the verification of the Non-Interactive-Auth message.
    - Extracts the Auth MAC from the Non-Interactive-Auth message and verifies that it is equal to the one just calculated. If it is not, ignore the Non-Interactive-Auth message.
6. Computes the Mixed shared secret and the SSID:
    - K = KDF(usage_shared_secret || tmp_k, 64). Securely deletes tmp_k and brace_key.
    - Calculates the SSID from shared secret: HWC(usage_SSID || K, 8).
7. Initializes the double ratchet algorithm:
    - Sets since_last_dh as 0.
    - Sets i, j, k and pn as 0.
    - Sets max_remote_i_seen as -1.
    - Calculates the root key and receiving chain key: curr_root_key = KDF(usage_root_key || K, 64) and chain_key_receiving[k] = KDF(usage_chain_key || K, 64).
    - Sets the received their_ecdh_first from Alice as their_ecdh.
    - Sets the received their_dh_first from Alice as their_dh.
8. At this point, the non-interactive DAKE is complete for Bob:
    - If he immediately receives a data message, he follows what is defined in the When you send a Data Message section. Note that he will not perform a new DH ratchet for this message, but rather use the already derived chain_key_receiving[k].
    - If he wants to send a data message, he follows what is defined in the When you send a Data Message section. Note that he will perform a new DH ratchet for this message.

Prekey Message

This message is created and published to an untrusted Prekey Server to allow offline conversations. Each Prekey message contains two one-time use ephemeral public prekey values.

A valid Prekey message is generated as follows:

1. Generate a unique random id that is going to act as an identifier for this prekey message. It should be 4 byte unsigned value, big-endian (INT).
2. Create the first one-time use prekey by generating the ephemeral ECDH key pair, as defined in Generating ECDH and DH Keys:
    - secret key y (57 bytes).
    - public key Y.
3. Create the second one-time use prekey by generating the ephemeral DH key pair, as defined in Generating ECDH and DH Keys:
    - secret key b (80 bytes).

    - public key `B`.
4. Use the same instance tag from the Client Profile and Prekey Profile's owner. Additional messages in this conversation will continue to use this tag as the sender's instance tag. Also, this tag is used to filter future received messages. Messages intended for this instance of the client will have this number as the receiver's instance tag.

To verify the Prekey message:

1. Verify if the message type is `0x0F`.
2. Verify that protocol's version of the message is `0x0004`.
3. Check that the ECDH public key `Y` is on curve Ed448. See Verifying that a point is on the curve section for details.
4. Verify that the DH public key `B` is from the correct group. See Verifying that an integer is in the DH group section for details.

A Prekey message is an OTRv4 message encoded as:

```
Protocol version (SHORT)
  The version number of this protocol is 0x0004.

Message type (BYTE)
  The message has type 0x0F.

Prekey Message Identifier (INT)
  A prekey message id used for local storage and retrieval.

Prekey owner's instance tag (INT)
  The instance tag of the client that created the prekey.

Y Prekey owner's ECDH public key (POINT)
  First one-time use prekey value.

B Prekey owner's DH public key (MPI)
  Second one-time use prekey value. The ephemeral public DH key. Note that even
  though this is in uppercase, this is NOT a POINT.
```

Non-Interactive-Auth Message

This message finishes the non-interactive DAKE. It contains a key-only unforgeable message authentication code function. Bob sends it to Alice to commit to a choice of his ECDH ephemeral key and his DH ephemeral key, and to acknowledge Alice's ECDH ephemeral key and DH ephemeral key.

A valid Non-Interactive-Auth message is generated as follows:

1. If there is not a valid Client Profile, create a Client Profile, as defined in the Creating a Client Profile section. Otherwise, use the one you have on local storage.
2. Generate an ephemeral ECDH key pair, as defined in Generating ECDH and DH Keys:
    - secret key `x` (57 bytes).
    - public key `X`.
3. Generate an ephemeral DH key pair, as defined in Generating ECDH and DH Keys:
    - secret key `a` (80 bytes).
    - public key `A`.
4. Generate another ephemeral ECDH key pair, as defined in Generating ECDH and DH keys:
    - secret key `our_ecdh_first.secret` (57 bytes).
    - public key `our_ecdh_first.public`.
5. Generate another ephemeral DH key pair, as defined in Generating ECDH and DH keys:
    - secret key `our_dh_first.secret` (80 bytes).
    - public key `our_dh_first.public`.
6. Compute `K_ecdh = ECDH(x, their_ecdh)`.
7. Compute `k_dh = DH(a, their_dh)` and `brace_key = KDF(usage_third_brace_key || k_dh, 32)`. Securely delete `k_dh`.
8. Compute `tmp_k = KDF(usage_tmp_key || K_ecdh || ECDH(x, their_shared_prekey) || ECDH(x, H_b) || brace_key, 64)`. Securely delete `K_ecdh`. This value is needed for the generation of the Mixed shared secret.
9. Calculate the Auth MAC key `auth_mac_k = KDF(usage_auth_MAC_key || tmp_k, 64)`.
10. Compute `t = HWC(usage_non_int_auth_bob_client_profile || Bob_Client_Profile, 64) || HWC(usage_non_int_auth_alice_client_profile || Alice_Client_Profile, 64) || Y || X || B || A || their_shared_prekey || HWC(usageNonIntAuthPh || phi, 64)`.
11. Compute `sigma = RSig(H_a, sk_ha, {F_b, H_a, Y}, t)`. Refer to the Ring Signature Authentication section for details.
12. Attach the 'Prekey Message Identifier' that is stated in the retrieved Prekey message.
13. Generate a 4-byte instance tag to use as the sender's instance tag. Additional messages in this conversation will continue to use this tag as the sender's instance tag. Also, this tag is used to filter future received messages. Messages intended for this instance of the client will have this number as the receiver's instance tag.

To verify a Non-Interactive-Auth message:

1. Compute t = HWC(usage_non_int_auth_bob_client_profile || Bobs_Client_Profile, 64) || HWC(usage_non_int_auth_alice_client_profile || Alices_Client_Profile, 64) || Y || X || B || A || our_shared_prekey.public || HWC(usage_non_int_auth_phi || phi, 64).
2. Verify sigma as defined in the Ring Signature Authentication section. As multiple client profiles can coexist on local storage for a short amount of time, if the sigma verification fails and there are others profiles in local storage, use them to generate t and validate sigma: RVrf({F_b, H_a, Y}, sigma, t).

A Non-Interactive-Auth is an OTRv4 message encoded as:

```
Protocol version (SHORT)
  The version number of this protocol is 0x0004.

Message type (BYTE)
  The message has type 0x0D.

Sender's instance tag (INT)
  The instance tag of the person sending this message.

Receiver's instance tag (INT)
  The instance tag of the intended recipient.

Sender's Client Profile (CLIENT-PROF)
  As described in the section "Creating a Client Profile".

X (POINT)
  The ephemeral public ECDH key.

A (MPI)
  The ephemeral public DH key. Note that even though this is in uppercase, this
  is NOT a POINT.

Sigma (RING-SIG)
  The 'RING-SIG' proof of authentication value.

Prekey Message Identifier (INT)
  The 'Prekey Message Identifier' from the Prekey message that was retrieved
  from the untrusted Prekey Server, as part of the Prekey Ensemble.

Auth MAC (MAC)
  The MAC with the appropriate MAC key (see above) of the message ('t') for the
  Ring Signature ('RING-SIG').

our_ecdh_first.public
  The ephemeral public ECDH key that will be used for the intialization of
  the double ratchet algorithm.

our_dh_first.public
  The ephemeral public DH key that will be used for the intialization of
  the double ratchet algorithm.
```

Publishing Prekey Ensembles

For starting a non-interactive conversation, an user must publish to an untrusted Prekey Server these values:

- A Client Profile (CLIENT-PROF)
- A Prekey Profile (PREKEY-PROF)
- A set of prekey messages

An user only needs to upload its Client Profile and Prekey Profile to the untrusted Prekey Server once for the long-term public key it locally has, until this profile expire or one of its values changes.

However, this party may upload new prekey messages at other times, as defined in the Publishing Prekey Messages section.

The party will also need to upload a new Client Profile and a new Prekey Profile when they expire. These new values replace the old ones. Take into account, however, that Client Profiles and Prekey Profiles will have an overlap period of extra validity, so they can be used when delayed encrypted offline messages arrive. After this extra validity time ends, they must be securely deleted from storage.

The combination of one Client Profile, one Prekey Profile and one Prekey message is called a "Prekey Ensemble".

Details on how to interact with an untrusted Prekey Server to publish these values are outside the scope of this protocol. They can be found in the OTRv4 Prekey Server Specification.

Publishing Prekey Messages

An OTRv4 client must generate a user's prekey messages and publish them to an untrusted Prekey Server. Implementers are expected to create their own policy dictating how often their clients upload prekey messages to the Prekey Server. Nevertheless, prekey messages should be published to the Prekey Server once the server store of prekeys messages gets low.

Validating Prekey Ensembles

Use the following checks to validate a Prekey Ensemble. If any of the checks fail, ignore the Prekey Ensemble:

1. Check that all the instance tags on the Prekey Ensemble's values are the same.
2. Validate the Client Profile.
3. Validate the Prekey Profile.
4. Check that the Prekey Profile is signed by the same long-term public key stated in the Client Profile.
5. Verify the Prekey message as stated in its section.
6. Check that the OTR version of the prekey message matches one of the versions signed in the Client Profile contained in the Prekey Ensemble.
7. Check if the Client Profile's version is supported by the receiver.

Note that these steps can be done in anticipation of sending a Non-Interactive-Auth message.

Receiving Prekey Ensembles

Details on how prekey ensembles may be received from an untrusted Prekey Server are outside the scope of this protocol. This specification assumes that none, one or more than one prekey ensembles may arrive. It also assumes that for every received Client Profile and Prekey Profile, at least, one prekey message might arrive. If the prekey server cannot return one of the three values needed for a Prekey Ensemble, the non-interactive DAKE must wait until this value can be obtained.

Note that when a prekey message is retrieved, it should be deleted from storage in the untrusted Prekey Server. Nevertheless, the Client Profile and the Prekey Profile should not be deleted until they are replaced when expired or when one of its values changed.

The following guide is meant to help implementers identify and remove invalid prekey ensembles.

If the untrusted Prekey Server cannot return one of the three values needed for a Prekey Ensemble (a Client Profile, a Prekey Profile and a Prekey message):

1. The non-interactive DAKE must wait until this value can be obtained.

If one Prekey Ensemble is received:

1. Validate the Prekey Ensemble.
2. If the Prekey Ensemble is valid, decide whether to send a Non-Interactive-Auth message depending on whether the long-term key in the Client Profile is trusted or not. This decision is optional.

If many prekey ensembles are received:

1. Validate the Prekey Ensembles.
2. Discard all invalid prekey ensembles.
3. Discard all duplicate prekey ensembles in the list.
4. If one Prekey Ensemble remains:
    - Decide whether to send a message using this Prekey Ensemble if the long-term key within the Client Profile is trusted or not. This decision is optional.
5. If multiple valid prekey ensembles remain:
    - If there are keys that are untrusted and trusted in the list of messages, decide whether to only use the trusted long-term keys; and send messages with each one of them. This decision is optional.
    - If there are several instance tags in the list of prekey ensembles, decide which instance tags to send messages to.
    - If there are multiple prekey ensembles per instance tag, decide whether to send multiple messages to the same instance tag.

KCI Attacks

One aspect of online deniability that is often overlooked is the relationship between DAKEs and key compromise impersonation attacks. A KCI attack begins when the long-term secret key of a party is compromised. With this long-term secret key, an attacker can impersonate other users to the owner of the key. The DAKEs used in OTRv4 are inherently vulnerable to KCI, but this can be a desirable property. However, OTRv4 includes forging keys to mitigate against this vulnerability.

In theory, a user who claims to cooperate with a judge may justifiably refuse to reveal their long-term secret key because it would make them vulnerable to a KCI attack. The design of the DAKEs used in OTRv4 makes it impossible for the user to provide proof of communication to a judge without revealing their long-term secret key. This prevents a judge and informant from devising a protocol

where the judge is given cryptographic proof of communication while the informant suffers no repercussions. However, this scenario seems to be mostly theoretical. The more common case in practice may be the one in which the judge has access to the party's long-term secret keys. In this latter case, a KCI vulnerability can be a desirable property.

To prevent an scenario where a judge has access to the party's long-term secret key and still make it impossible for this party to provide proof of communication, we can use two alternatives:

1. In the case of the non-interactive DAKE, ask the Prekey Server for a forged conversation.
2. Include long-term "forger" keys in the DAKEs for both participants. This is the choice that OTRv4 has chosen.

Prekey Server Forged Conversations

The security of the non-interactive DAKE does not require trusting the prekey server used to distribute prekeys ensembles. However, if we allow a scenario in which one party's long-term keys have been compromised but the prekey server has not, we can achieve better deniability properties. The party may ask the prekey server in advance for a forged conversation, which will cast doubt on all conversations conducted by an attacker using the compromised device.

If an attacker attempts to act as Bob in the above overview of the non-interactive DAKE using a compromised device, then he (or a trusted accomplice with access to Bob's long-term secret key) can impersonate Alice by executing RSig with his long-term public and secret keys ($\mathrm{sigma = RSig(Hb, sk\_hb, \{Ha, Hb, Y\}, t)}$ instead. In practice, Bob (or his accomplice) simply needs to run the non-interactive DAKE honestly, but pretend to be Alice in their response to the prekey message.

If an attacker attempts to act as Alice in the above non-interactive DAKE overview using the compromised device, we cannot offer full deniability. Alice must ask the prekey server to return a false prekey ensemble from Bob that was generated by Bob or his trusted accomplice, and to redirect all traffic to the associated forging device. This false prekey ensemble must be returned to the attacker when they request one. Alice can derive the Mixed shared secret by using Bob's long-term public key instead of hers. In practice, the attacker can always bypass this forgery attempt by obtaining a legitimate prekey ensemble from Bob and using this to respond. This limitation of the non-interactive DAKE is conjectured to be insurmountable by a two-flow non-interactive protocol [13].

Forger Keys

For OTRv4, the above KCI vulnerability is undesirable. For this reason, the protocol design includes 'forging keys', which should be generated and implemented. To use them, both DAKEs are altered to include these long-term forging keys for all participants. In the case of the non-interactive DAKE, for example, Bob distributes these keys on his Client Profile. Alice, after retrieving the Prekey Ensemble, extracts the 'OTRv4 public Ed448 forging key' ($\mathrm{F\_b}$) and uses to compute $\mathrm{sigma = RSig(H\_a, sk\_ha, \{F\_b, Ha, Y\}, t)}$.

This transformation changes all long-term public keys in the protocol that are not used in the "honest" case to reference to the forging keys instead. This alteration allows the forging keys to be stored more securely than the "honest" long-term public keys; since the forging keys are not needed for normal operation, they may be stored offline. Additionally, long-term forging keys don't need to be generated like the "honest" long-term public keys; they can be a random valid point on the curve, as described in Public keys, Shared Prekeys, Forging keys and Fingerprints section. Alternatively, the forging secret keys can be destroyed immediately after generation. This last technique have to be implemented as an option for users: a secure messaging application should ask users whether or not they would like to save the forging keys during setup. Even if most users select the default option to securely erase the forging keys, thereby preventing them from performing the online forgery techniques described in the section above, someone watching the protocol execution does not generally know the choice of a particular user. Consequently, a judge that engages in a conversation using a compromised device is given two explanations: either the conversation is genuine, or the owner of the device was one of the users that elected to store the forgery keys and they are using those keys to forge the conversation.

Note that forging keys have to be included in the generation of the fingerprint as well, as this will prevent an attack where a judge forces a participant (Bob, for example) to use specific keys as the forging long-term key and the ephemeral DAKE key, and advertise this new forging key in a new published Client Profile. When Alice performs a DAKE with Bob, she sends Bob her ring signature (in the 'Auth-R' message, for example). Since the judge knows the private keys associated with the coerced keys (the forger and ephemeral one), the judge learns that the signature is from Alice (as only she could have made it), which generates proof of Alice participation. In this case, Bob does not suffer any repercussions as he can simply let the compromised Client Profile expire and update his forging key to other value that he generated himself. To prevent this attack, Alice should always verify that, when performing a DAKE, only the keys that she trusts (because she has done a manual fingerprint verification or executed the Socialist Millionaires Protocol with Bob) are the ones actually used, so it is not that easy for a judge to coerce Bob to change his long-term forging keys without raising suspicion. Nevertheless, a judge can always try to force Bob to do a manual fingerprint verification of the coerced keys with Alice. This will likely trick her into believing that those are Bob's non-coerced keys. This latter scenario seems difficult to perform without raising suspicion.

Data Exchange

This section describes how each participant will use the Double Ratchet algorithm to exchange Data Messages. The Double Ratchet algorithm is initialized with the shared secret established in the DAKE and the public keys immediately exchanged. Detailed validation and

processing of each data message is described in the sending a Data Message and receiving a Data Messages sections.

A data message with an empty human-readable part (the plaintext is of zero length, or starts with a NULL) is a "heartbeat" message. This message is useful for key rotations and revealing MAC keys. It should not be displayed to the participant. If you have not sent a message to a correspondent in some (configurable) time, send a "heartbeat" message. The "heartbeat" message should have the IGNORE_UNREADABLE flag set.

```
Alice                                               Bob
-------------------------------------------------------------------------------
Initialize root key, chain keys                 Initialize root key, chain keys
and the other party ECDH and DH keys                and the other party ECDH and DH keys

Derive MKenc & MKmac
Generate MAC,
Encrypt message 0

Send data message 0          ------------------->

Derive MKenc & MKmac
Generate MAC,
Encrypt message 1

Send data message 1          ------------------->

                                    Receive data message 0
                                    Compute receiving chain key 0
                                    Derive MKenc & MKmac
                                    Verify MAC, Decrypt message 0

                                    Receive data message 1
                                    Compute receiving chain key 1
                                    Derive MKenc & MKmac
                                    Verify MAC, Decrypt message 1

                          Perform a new DH Ratchet

                                    Derive MKenc & MKmac
                                    Generate MAC,
                                    Encrypt message 0

                          <------------------- Send data message 0

                                    Derive MKenc & MKmac
                                    Generate MAC,
                                    Encrypt message 1

                          <------------------- Send data message 1

Receive data message 0
Compute receiving chain key 0
Derive MKenc & MKmac
Verify MAC, Decrypt message 0

Receive data message 1
Compute receiving chain key 1
Derive MKenc & MKmac
Verify MAC, Decrypt message 1
```

Data Message

This message is used to transmit a private message to the correspondent. It is also used to Reveal Old MAC Keys. This data message is encoded as defined in the Encoded Messages section.

The plaintext message (either before encryption or after decryption) consists of a human-readable message (encoded in UTF-8, optionally with HTML markup), optionally followed by:

- a single NUL (a BYTE with value 0x00)
- zero or more TLV (type/length/value) records (with no padding between them)

Data Message Format

```
Protocol version (SHORT)
  The version number of this protocol is 0x0004.

Message type (BYTE)
```

```
  The Data Message has type 0x03.

Sender's instance tag (INT)
  The instance tag of the person sending this message.

Receiver's instance tag (INT)
  The instance tag of the intended recipient.

Flags (BYTE)
  The bitwise-OR of the flags for this message. Usually you should set this to
  0x00. The only currently defined flag is:

  IGNORE_UNREADABLE (0x01)

    If you receive a Data Message with this flag set, and you are unable to
    decrypt the message or verify the MAC (because, for example, you don't have
    the right keys), just ignore the message instead of producing an error or a
    notification to the participant.

Previous chain message number (INT)
  This should be set with sender's pn.

Ratchet id (INT)
  This should be set with sender's i.

Message id (INT)
  This should be set with sender's j.

Public ECDH Key (POINT)
  This is the public part of the ECDH key pair. For the sender of this message,
  this is their 'our_ecdh.public' value. For the receiver of this message, it is
  used as 'their_ecdh'.

Public DH Key (MPI)
  This is the public part of the DH key pair. For the sender of this message, it
  is 'our_dh.public' value. For the receiver of this message, it is used as
  'their_dh'. If this value is empty, its length is zero.

Encrypted message (DATA)
  Using the appropriate encryption key (see below) derived from the sender's
  and recipient's ECDH and DH public keys (with the keyids given in this
  message), perform an encryption of the message using ChaCha20. The 'nonce'
  used for this operation is set to 0.

Authenticator (MAC)
  The MAC with the appropriate MAC key (see below) of everything: from the
  protocol version to the end of the encrypted message. Note that old MAC keys
  are not included in this field.

Old MAC keys to be revealed (DATA)
  See "Revealing MAC Keys" section. This corresponds to the 'mac_keys_to_reveal'
  variable.
```

When you send a Data Message:

In order to send a data message, a key is required to encrypt the message in it. This per-message key (MKenc) is the output key from the sending and receiving KDF chains. As defined in [2], the KDF keys for these chains are called 'chain keys'. When a participant wants to send a data message after receiving another one, ratchet keys should be rotated (the ECDH keys, the brace key, the root key and the sending chain key) and the j parameter should be set to 0. This latter process is called a DH Ratchet.

Given a new DH Ratchet:

- Rotate the ECDH keys and brace key, see Rotating ECDH Keys and Brace Key as sender section. The new ECDH public key created by the sender in this process will be the 'Public ECDH Key' for the message. If a new public DH key is created in this process, it will be the 'Public DH Key' for the message. If it is not created (meaning it is only a KDF of the previous one), then it will be empty.
- Calculate the Mixed shared secret K = KDF(usage_shared_secret || K_ecdh || brace_key, 64). Securely deletes K_ecdh.
- Interpret curr_root_key as prev_root_key, if present.
- Derive new set of keys: curr_root_key, chain_key_s[j] = derive_ratchet_keys(sending, prev_root_key, K).
- Securely delete the previous root key (prev_root_key) and K.
- If present, forget and reveal MAC keys. The conditions for revealing MAC keys are stated in the Revealing MAC Keys section.
- Derive the next sending chain key, MKenc and MKmac, and encrypt the message as described below.

When sending a data message in the same DH Ratchet:

- Set i - 1 as the Data message's ratchet id.

- Set `j` as the Data message's message id.
- Set `pn` as the Data message's previous chain message number.
- Derive the next sending chain key `chain_key_s[j+1] = KDF(usage_next_chain_key || chain_key_s[j], 64)`.
- Calculate the encryption key (`MKenc`), the MAC key (`MKmac`) and, if needed the extra symmetric key:

```
MKenc, MKmac = derive_enc_mac_keys(chain_key_s[j])
extra_symm_key = KDF(usage_extra_symm_key || 0xFF || chain_key_s[j], 64)
```

- Securely delete `chain_key_s[j]`.
- Set `nonce` to 0.
- Use only the first 32 bytes of `MKenc` to encrypt the message:

```
encrypted_message = Chacha20_Enc(MKenc, nonce, m)
```

- Use the `MKmac` to create a MAC tag. MAC all the sections of the data message from the protocol version to the encrypted message.

```
Authenticator = HWMAC(usage_authenticator || MKmac || data_message_sections, 64)
```

- Increment the next sending message id `j = j + 1`.
- Securely delete `MKenc` and `MKmac`.
- Continue to use the sender's instance tag.

When you receive a Data Message:

The counterpart of the sending an encoded data message. As that one, it also needs a per-message key derived from the previous chain key to decrypt the message. If the receiving 'Public ECDH Key' has not yet been seen, ratchet keys should be rotated (the ECDH keys, the brace key, the root key and the receiving chain key). It also checks for duplicated messages and discards them.

Decrypting a data message consists of:

1. If the received encrypted message corresponds to an stored message key corresponding to an skipped message, the message is verified and decrypted with that key. Once used, that key is deleted from the storage.
2. If a new DH ratchet key is received, any message keys corresponding to skipped messages from the previous receiving DH ratchet are stored. A new DH ratchet is performed. The message is then verified and decrypted.
3. If a new message from the current receiving ratchet is received, any message keys corresponding to skipped messages from the same ratchet are stored, and a symmetric-key ratchet is performed to derive the current message key and the next receiving chain key. The message is then verified and decrypted.

If an error is raised or something fails (e.g. the MAC verification of the message fails), the message should be discarded, and any changes to the state variables or key variables should be discarded as well.

The decryption mechanism works as:

- Check that the receiver's instance tag matches your sender's instance tag.
    - If they do not match, discard the message.
- Try to decrypt the message with a stored skipped message key:
    - If the received `message_id` and `Public ECDH Key` are in the `skipped_MKenc` dictionary:
    - Get the message key and the extra symmetric key (if needed): `MKenc, extra_symm_key = skipped_MKenc[Public ECDH Key, message_id]`.
    - Calculate `MKmac = KDF(usage_MAC_key || MKenc, 64)`.
    - Use the `MKmac` to verify the MAC of the data message. If this verification fails:
        - Reject the message.
    - Securely delete `skipped_MKenc[Public ECDH Key, message_id]`.
    - Set `nonce` to 0.
    - Decrypt the message by using the nonce and only the 32 bytes of `MKenc`:

```
decrypted_message = ChaCha20_Dec(MKenc, nonce, m)
```

- Securely delete MKenc.
- Add MKmac to the list mac_keys_to_reveal.
- If max_remote_i_seen > ratchet_id:
- If the received message_id and Public ECDH Key are not in the skipped_MKenc dictionary:
  - This is a duplicated message from a past DH ratchet. Discard the message.
- If the received 'Public ECDH Key' is the same as their_ecdh and the 'Public DH Key' is the same as their_dh, and the keys were not found in the skipped_MKenc dictionary:
- If message_id < k:
  - This is a duplicated message. Discard the message.

- Given a new ratchet (the 'Public ECDH Key' is different from their_ecdh and the 'Public DH Key' is different from their_dh):
  - Store any message keys from the previous DH Ratchet that correspond to messages that have not yet arrived:
    - If k + max_skip < received Previous chain message number:
      - Raise an exception that informs the participant that too many message keys are stored.
    - If chain_key_r is not NULL:
      - while k < received Previous chain message number:
        - Derive chain_key_r[k+1] = KDF(usage_next_chain_key || chain_key_r[k], 64) and MKenc = KDF(usage_message_key || chain_key_r[k], 64)
        - Derive (this is done any time a message key is stored as there is no way of knowing if the message that will be received in the future will ask for the computation of the extra symmetric key): extra_symm_key = KDF(usage_extra_symm_key || 0xFF || chain_key_r[k], 64).
        - Store MKenc, extra_sym_key = skipped_MKenc[their_ecdh, k].
        - Increment k = k + 1.
        - Delete chain_key_r[k].
  - Rotate the ECDH keys and brace key, see Rotating ECDH Keys and Brace Key as receiver section.
  - Set pn as j.
  - Set j as 0.
  - Set k as 0.
  - Calculate K = KDF(usage_shared_secret || K_ecdh || brace_key, 64). Securely delete K_ecdh.
  - Interpret curr_root_key as prev_root_key, if present.
  - Derive new set of keys curr_root_key, chain_key_r[k] = derive_ratchet_keys(receiving, prev_root_key, K).
  - Securely delete the previous root key (prev_root_key) and K.
  - Derive the next receiving chain key, MKenc and MKmac, and decrypt the message as described below.

- When receiving a data message in the same DH Ratchet:
  - Store any message keys from the current DH Ratchet that correspond to messages that have not yet arrived:
  - If k + max_skip < received message_id:
    - Abort the decryption of that data message.
  - If chain_key_r is not NULL:
    - while k < received message_id:
    - Derive chain_key_r[k+1] = KDF(usage_next_chain_key || chain_key_r[k], 64) and MKenc = KDF(usage_message_key || chain_key_r[k], 64)
    - Derive (this is done any time a message key is stored as there is no way of knowing if the message that will be received in the future will ask for the computation of the extra symmetric key): extra_symm_key = KDF(usage_extra_symm_key || 0xFF || chain_key_r[k], 64).
    - Store MKenc, extra_sym_key = skipped_MKenc[their_ecdh, k].
    - Increment k = k + 1.
    - Delete chain_key_r[k].
  - Calculate the encryption and MAC keys (MKenc and MKmac).

```
MKenc, MKmac = derive_enc_mac_keys(chain_key_r[k])
extra_symm_key = KDF(usage_extra_symm_key || 0xFF || chain_key_r[k], 64)
```

- Use the MKmac to verify the MAC of the message. If the verification fails:
  - Reject the message.
  - Delete the derived MKenc and MKmac.
- Otherwise:
  - Derive the next receiving chain key: chain_key_r[k+1] = KDF(usage_next_chain_key || chain_key_r[k], 64).
  - Securely delete chain_key_r[k].
  - Increment the receiving message id k = k + 1.
  - Set nonce to 0.
  - Decrypt the message by using the nonce and only the 32 bytes of MKenc:

```
decrypted_message = ChaCha20_Dec(MKenc, nonce, m)
```

  - If the message cannot be decrypted:
  - Reject the message.
  - Securely delete MKenc.
  - Set their_ecdh as the 'Public ECDH key' from the message.
  - Set their_dh as the 'Public DH Key' from the message, if it is not empty.
  - Add MKmac to the list mac_keys_to_reveal.
- Set max_remote_i_seen to ratchet_id.

- If a message arrives that corresponds to a message key already deleted or that cannot be derived:
  - Reject the message.

Deletion of Stored Message Keys

Storing message keys from messages that haven't arrived yet introduces some risks, as defined in [2]:

1. A malicious sender could induce receivers to store large numbers of skipped message keys, possibly causing a denial-of-service due to consuming storage space.
2. An adversary can capture and drop some messages from sender, even though they didn't reach the recipient. The attacker can later compromise the intended recipient at a later time to reveal the stored message keys that correspond to the dropped messages. The adversary can then retroactively decrypt the captured messages.

To mitigate the first risk, parties should set reasonable per-conversation limits on the number of possible stored message keys (e.g. 1000). This limit is set by the implementers.

To mitigate the second risk, parties should delete stored message keys after an appropriate interval. This deletion could be triggered by a timer, or by counting the number of events (messages received, DH ratchet steps, etc.). This should be decided by the implementer. This partially defends against the second risk as it only protects "lost" messages, not messages sent using a new DH ratchet key that has not yet been received by the compromised party. To also defend against the second risk, the session should be regularly expired, as defined in the Session Expiration section.

Extra Symmetric Key

Like OTRv3, OTRv4 defines an additional symmetric key that can be derived by the communicating parties for use of application-specific purposes, such as file transfer, voice encryption, etc. When one party wishes to use the extra symmetric key, they create a type 7 TLV, which they attach to a Data Message. The extra symmetric key itself is then derived using the same chain_key used to compute the message encryption key used to protect the Data Message. It is, therefore, derived by calculating KDF(usage_extra_symm_key || 0xFF || chain_key).

Upon receipt of the Data Message containing the type 7 TLV, the recipient will compute the extra symmetric key in the same way. Note that the value of the extra symmetric key is not contained in the TLV itself.

If more keys are wished to be derived from this already calculated extra symmetric key, this can be done by taking the index from the TLV list received in the data message and the context received in 7 TLV (the 4-byte indication of what this symmetric key will be used for), and use them as inputs to a KDF:

```
symkey1 = KDF(index || context || extra_sym_key, 64)
```

So, if for example, these TLVs arrive with the data message:

```
TLV 1
TLV 7   context: 0x0042
TLV 2
TLV 7   context: 0x104A
TLV 3
TLV 7   context: 0x0001
```

Three keys can, therefore, be calculated from the already derived extra symmetric key:

```
extra_sym_key = KDF(usage_extra_symm_key || 0xFF || chain_key, 64)
symkey1 = KDF(0x00 || 0x0042 || extra_sym_key, 64)
symkey2 = KDF(0x01 || 0x104A || extra_sym_key, 64)
symkey3 = KDF(0x02 || 0x0001 || extra_sym_key, 64)
```

Every derived key and the `extra_symm_key` should be deleted after being used.

Revealing MAC Keys

Old MAC keys are keys from already received messages, that will no longer be used to verify the authenticity of that message. We reveal them in order to provide Forgeability of Messages: once MAC keys are revealed, anyone can modify an OTR message and still have it appear as valid.

A MAC key is added to `mac_keys_to_reveal` list after a participant has verified the message associated with that MAC key. They are also added if the session is expired or when the storage of message keys gets deleted, and the MAC keys for messages that have not arrived are derived.

Old MAC keys are formatted as a list of 64-byte concatenated values. The first data message sent every ratchet reveals them or the TLV type 1 that is used when the session is expired.

Fragmentation

Some networks may have a maximum message size that is too small to contain an encoded OTR message. In that event, the sender may choose to split the message into a number of fragments. This section describes the format for the fragments.

OTRv4 fragmentation and reassembly procedure needs to be able to break OTR messages into an almost arbitrary number of pieces that can be later reassembled. The receiver of the fragments uses the identifier field to ensure that fragments of different messages are not mixed. The fragment index field tells the receiver the position of a fragment in the original data message. These fields provide sufficient information to reassemble data messages.

OTRv4 and OTRv3 perform fragmentation in different ways. As OTRv4 supports an out-of-order network model, fragmentation is different. Nevertheless, for both OTR versions, message parsing should happen after the message has been defragmented.

All OTRv4 clients must be able to reassemble received fragments, but performing fragmentation on outgoing messages is optional.

For fragmentation in OTRv3, refer to the "Fragmentation" section on OTRv3 specification.

Transmitting Fragments

If you have information about the maximum message size you are able to send (different IM networks have different limits), you can fragment an encoded OTR message as follows:

- Start with the OTR message as you would normally transmit it. For example, a Data Message would start with `?OTR:AAQD` and end with `.`.
- Assign an identifier, which will be used specifically for this fragmented data message. This is done in order to not confuse these fragments with other data message's fragments. The identifier is a unique randomly generated 4-byte value that must be unique for the time the data message is fragmented.
- Break it up into sufficiently small pieces. Let this number of pieces be `total`, and the pieces be `piece[1]`,`piece[2]`,...,`piece[total]`.
- Transmit `total` OTRv4 fragmented messages with the following (printf-like) structure (as `index` runs from 1 to `total` inclusive:

  ```
  "?OTR|%x|%x|%x,%hu,%hu,%s,", identifier, sender_instance, receiver_instance, index, total, piece[index]
  ```

OTRv3 messages get fragmented in a similar format, but without the identifier field:

```
"?OTR|%x|%x,%hu,%hu,%s,", sender_instance, receiver_instance, index, total, piece[index]
```

The message should begin with `?OTR|` and end with `,`.

Note that `index` and `total` are unsigned short int (2 bytes), and each has a maximum value of 65535. Each `piece[index]` must be non-empty. The `identifier`, instance tags, `index` and `total` values may have leading zeros.

Note that fragments are not messages that can be fragmented: you can't fragment a fragment.

Receiving Fragments

A reassemble process does not to be implemented in precesely the way we are going to describe; but the process implemented in a library has to be able to correctly reassemble the fragments.

If you receive a message containing `?OTR|` (note that you'll need to check for this before checking for any of the other `?OTR:` markers):

- Parse it (as the previous printf structure) extracting the `identifier`, the instance tags, `index`, `total`, and `piece[index]`.
- Discard the message and optionally pass a warning to the participant if:
  - The recipient's own instance tag does not match the listed receiver instance tag, and
  - The listed receiver's instance tag is not zero.
- Discard the (illegal) fragment if:
  - `index` is 0
  - `total` is 0
  - `index` is bigger than `total`
- For the first fragment that arrives (there is not a current buffer with the same `identifier`):
  - Create a buffer which will keep track of the portions of the fragmented data message that have arrived (by filling up it with fragments).
  - Optionally, initialize a timer for the reassembly of the fragments as it is possible that some fragments of the data message might never show up. This timer ensures that a client will not be "forever" waiting for a fragment. If the timer runs out, all stored fragments in this buffer should be discarded.
  - Let `B` be the buffer, `I` be the currently stored identifier, `T` the currently stored `total` and `C` a counter that keeps track of the received number of fragments for this buffer. If you have no currently stored fragments, there are no buffers, and `I`, `T` and `C` equal 0.
  - Set the length of the buffer as `total`: `len(B) = total`.
  - If `index` is empty, store `piece` at the `index` given position: `insert(piece, index)`. If it is not, reject the fragment and do not increment the buffer counter.
  - Let `total` be `T` and `identifier` be `I` for the buffer.
  - Increment the buffer counter: `C = C + 1`.
- If `identifier == I`:
  - If `total == T`, and `C < T`:
    - Check that the given position of the buffer is empty: `B[index] == NULL`. If it is not, reject the fragment and do not increment the buffer counter.
    - Store the `piece` at the given position in the buffer: `insert(piece, index)`.
    - Increment the buffer counter: `C = C + 1`.
  - Otherwise:
    - Forget any stored fragments of this buffer you may have.
    - Reset `C` and `I` to 0, and discard this buffer.
- Otherwise:
  - Consider this fragment as part of another buffer: either create a new buffer or insert the fragment into one that has already been created.

After this, if the current buffer's `C == T`, treat the buffer as the received data message.

If you receive a non-OTR message or an unfragmented message:

- Keep track of the buffers you may already have. Do not discard them.

For example, here is a Data Message we would like to transmit over a network with an unreasonably small maximum message size:

```
?OTR:AAMDJ+MVmSfjFZcAAAAAAQAAAAIAAADA1g5IjD1ZGLDVQEyCgCyn9hb
rL3KAbGDdzE2ZkMyTKl7XfkSxh8YJnudstiB74i4BzT0W2haClg6dMary/jo
9sMudwmUdlnKpIGEKXWdvJKT+hQ26h9nzMgEditLB8vjPEWAJ6gBXvZrY6ZQ
rx3gb4v0UaSMOMiR5sB7Eaulb2Yc6RmRnnlxgUUC2alosg4WIeFN951PLjSc
ajVba6dqlDi+q1H5tPvI5SWMN7PCBWIJ41+WvF+5IAZzQZYgNaVLbAAAAAAA
AAAEAAAAHwNiIi5Ms+4PsY/L2ipkTtquknfx6HodLvk3RAAAAAA==.
```

We could fragment this message into three pieces:

```
?OTR|3c5b5f03|5a73a599|27e31597,00001,00003,?OTR:AAMDJ+MVmSfjFZcAAAAA
AQAAAAIAAADA1g5IjD1ZGLDVQEyCgCyn9hbrL3KAbGDdzE2ZkMyTKl7XfkSx
h8YJnudstiB74i4BzT0W2haClg6dMary/jo9sMudwmUdlnKpIGEKXWdvJKT+
hQ26h9nzMgEditLB8v,

?OTR|3c5b5f03|5a73a599|27e31597,00002,00003,jPEWAJ6gBXvZrY6ZQrx3gb4v0
UaSMOMiR5sB7Eaulb2Yc6RmRnnlxgUUC2alosg4WIeFN951PLjScajVba6dq
lDi+q1H5tPvI5SWMN7PCBWIJ41+WvF+5IAZzQZYgNaVLbAAAAAAAAAAEAAAA
HwNiIi5Ms+4PsY/L2i,

?OTR|3c5b5f03|5a73a599|27e31597,00003,00003,pkTtquknfx6HodLvk3RAAAAAA
==.,
```

The Protocol State Machine

An OTR client maintains separate state for every correspondent. For example, Alice may have an active OTR conversation with Bob, while having an insecure conversation with Charlie.

The way the client reacts to user input and to received messages depends on whether the client has decided to allow version 3 and/or 4, if encryption is required and if it will advertise OTR support.

Protocol States

```
START

  This is the initial state before an OTRv4 or OTRv3 conversation starts. The
  only way to enter this state is for the participant to explicitly request it
  via some UI operation. Messages sent in this state are plaintext messages. If
  a TLV type 1 (Disconnected) message is sent in ENCRYPTED_MESSAGES state,
  transition to this state (except when the session is expired). Note that this
  transition only happens when TLV type 1 message is sent, not when it is
  received.

WAITING_AUTH_R

  This is the state used when a participant is waiting for an Auth-R message.
  This state is entered after an Identity message is sent.

WAITING_AUTH_I

  This is the state used when a participant is waiting for an Auth-I message.
  This state is entered after sending an Auth-R message.

ENCRYPTED_MESSAGES

  This state is entered after the DAKE is finished. The interactive DAKE is
  finished, for Bob, after the Auth-I message is sent, and, for Alice, when the
  Auth-I message is received and validated. The non-interactive DAKE is
  finished, for Alice, when the Non-Interactive-Auth message is sent, and, for
  Bob, when the Non-Interactive-Auth message is received and validated. Outgoing
  messages sent in this state are encrypted. Query messages or plaintext with
  withespace tags are not allowed to be sent in this state.

FINISHED

  This state is entered only when a participant receives a TLV type 1
  (Disconnected) message, which indicates they have terminated their side
  of the OTRv4 conversation. For example, if Alice and Bob are having an OTRv4
  conversation, and Bob instructs his OTRv4 client to end its private session
```

```
with Alice (for example, by logging out), Alice will be notified of this,
and her client will switch to the FINISHED state. This prevents Alice from
accidentally sending a message to Bob in plaintext (consider what happens
if Alice was in the middle of typing a private message to Bob when he
suddenly closes the session, just as Alice hits the 'enter' key). Note that
this transition only happens when TLV type 1 message from OTRv4 is received,
not when it is sent. This state indicates that outgoing messages are not
delivered at all. If a OTRv3 message is received in this state, it should be
ignored.
```

Protocol Events

The following sections outline the actions that the protocol should implement. This assumes that the client is initialized with the allowed versions (3 and/or 4).

There are thirteen events an OTRv4 client must handle (for version 3 messages, please refer to the previous OTR protocol document. The only state where OTRv3 messages are taken into account is the START state):

- Received messages:
    - Plaintext without the whitespace tag
    - Plaintext with the whitespace tag
    - Query Messages
    - Error Message
    - Identity Message
    - Auth-R Message
    - Auth-I Message
    - Non-Interactive-Auth Message
    - Data Message
- User actions:
    - User requests to start an OTR conversation
    - Starting a conversation interactively
    - User requests to end an OTR conversation
    - Sending an encrypted data message

For version 4 messages, someone receiving a message with a recipient instance tag specified that does not equal their own, should discard the message and optionally warn the user. The exception here is the Identity Message where the receiver's instance tag may be 0, indicating that no particular instance is specified, and the Prekey Ensemble, whose values do not include this field.

User requests to start an OTR Conversation

Send an OTR Query Message or a plaintext message with a whitespace tag to the correspondent. Query Messages and Whitespace Tags are constructed according to the sections below.

Query Messages

If Alice wishes to communicate to Bob that she would like to use OTR, she sends a message containing the string "?OTRv" followed by an indication of what versions of OTR she is willing to use with Bob. The versions she is willing to use, whether she can set this on a global level or per-correspondent basis, is up to the implementer. However, enabling users to choose whether they want to allow or disallow a version is required, as OTR clients can set different policies for different correspondents. For example, Alice could set up her client so that it speaks only OTR version 4, except with Charlie, who she knows has only an old client; so that it will opportunistically start an OTR conversation whenever it detects the correspondent supports it; or so that it refuses to send non-encrypted messages to Bob, ever.

Note that query Messages are not allowed to be sent in ENCRYPTED_MESSAGES state.

The version string is constructed as follows:

If Alice is willing to use OTR, she appends a byte identifier for the versions in question, followed by "?". The byte identifier for OTR version 3 is "3", and "4" for 4. Thus, if she is willing to use OTR versions 3 and 4, the identifier would be "34". The order of the identifiers between the "v" and the "?" does not matter, but none should be listed more than once. The OTRv4 specification only supports versions 3 and higher. Thus, query messages for older versions have been omitted.

Example query messages:

```
"?OTRv3?"
```

```
    Version 3
"?OTRv45x?"
    Version 4, and hypothetical future versions identified by "5" and "x"
"?OTRv?"
    A bizarre claim that Alice would like to start an OTR conversation, but is
    unwilling to speak any version of the protocol. Although this is
    syntactically valid, the receiver will not reply when receiving this.
```

These strings may be hidden from the user (for example, in an attribute of an HTML tag), and may be accompanied by an explanatory message which should not reveal information regarding the participants (an example can be "Your buddy has requested an Off-the-Record private conversation."). If Bob is willing to use OTR with Alice (with a protocol version that Alice has offered), he should start the AKE or DAKE according to the compatible version he supports.

Whitespace Tags

If Alice wishes to communicate to Bob that she is willing to use OTR, she can attach a special whitespace tag to any plaintext message she sends him. A Whitespace tag may occur anywhere in the message, and may be hidden from the user (as in the Query Messages). There should be only one whitespace tag per message. In the case that multiple whitespace tags arrive, only the first one should be considered as valid.

The tag consists of the following 16 bytes, followed by one or more sets of 8 bytes indicating the version of OTR Alice is willing to use:

```
Always send "\x20\x09\x20\x20\x09\x09\x09\x09"
"\x20\x09\x20\x09\x20\x09\x20\x20",
followed by one or more of:
  "\x20\x20\x09\x09\x20\x20\x09\x09"
to indicate a willingness to use OTR version 3 with Bob or
  "\x20\x20\x09\x09\x20\x09\x20\x20"
to indicate a willingness to use OTR version 4 with Bob
```

If Bob is willing to use OTR with Alice, with the protocol version that Alice has offered, he should start the AKE or DAKE. On the other hand, if Alice receives a plaintext message from Bob (rather than an initiation of the AKE or DAKE), she should stop sending him a whitespace tag.

Receiving plaintext without the whitespace tag

Display the message to the user. Depending on the policy set up by the client or the mode in which it was initiated, the user should be warned that the message received was unencrypted.

If the state is ENCRYPTED_MESSAGES or FINISHED:

- Display the message to the user, depending on the policy set up by the client or the mode in which it was initiated. The user should be warned that the message received was unencrypted.

For OTRv3, if msgstate is MSGSTATE_ENCRYPTED or MSGSTATE_FINISHED:

- Display the message to the user. The user should be warned that the message received was unencrypted.

Receiving plaintext with the whitespace tag

Remove the whitespace tag and display the message to the user. Depending on the policy set up by the client or the mode in which it was initiated, the user should be warned that the message received was unencrypted.

If the state is ENCRYPTED_MESSAGES or FINISHED:

- Remove the whitespace tag and display the message to the user. The user should be warned that the message received was unencrypted.

For OTRv3, if msgstate is MSGSTATE_ENCRYPTED or MSGSTATE_FINISHED:

- Display the message to the user. The user should be warned that the message received was unencrypted.

In any event:

- If the tag offers OTR version 4 and version 4 is allowed:
    - Send an Identity message.
    - Transition the state to WAITING_AUTH_R.

- If the tag offers OTR version 3 and version 3 is allowed:
    - Send a version 3 D-H Commit Message.
    - Transition authstate to AUTHSTATE_AWAITING_DHKEY.

Sending a Query Message after an offline conversation

In the case that a party received offline messages, comes online and wants to send online messages:

- Send a TLV type 1 (Disconnected).
- Send a Query Message.

Receiving a Query Message

If the Query Message offers OTR version 4 and version 4 is allowed:

- Send an Identity message.
- Transition the state to WAITING_AUTH_R.

If the Query message offers OTR version 3 and version 3 is allowed:

- Send a version 3 D-H Commit Message.
- Transition authstate to AUTHSTATE_AWAITING_DHKEY.

Starting a conversation interactively

Rather than requesting to start an encrypted conversation, Alice can directly start a OTRv4 conversation with Bob if she is certain that they both support it and are willing to do so. In such case, Alice should:

- Send an Identity message.
- Transition the state to WAITING_AUTH_R.

For how to start a conversation interactively, check the modes folder, either the OTRv4-interactive-only mode or the OTRv4-standalone-mode one.

Receiving an Identity Message

If the state is START:

- Validate the Identity message. Ignore the message if validation fails. Note that after receiving an Indentity message, a participant must not start sending data messages.
- If validation succeeds:
    - Remember the sender's instance tag to use as the receiver's instance tag for future messages.
    - Reply with an Auth-R message.
    - Transition to the WAITING_AUTH_I state.

If the state is WAITING_AUTH_R:

You and the other participant have sent Identity messages to each other. This can happen if they send you an Identity message before receiving yours. Only one Identity message must be chosen for use.

- Validate the Identity message. Ignore the message if validation fails.
- If validation succeeds:
    - Compare the hashed B you sent in your Identity message with the DH value from the message you received, considered as 32-byte unsigned big-endian values.
    - If yours is the higher hash value:
        - Ignore the incoming Identity message, but resend your Identity message. This means that the other side have the lower hash value and, therefore, will keep going as stated below.
    - Otherwise:
        - Forget the old our_ecdh, our_dh, our_ecdh_first.public and our_dh_first.public values that you sent earlier.
        - Pretend you are on START state.
        - Send a new Auth-R message.
        - Transition state to WAITING_AUTH_I.

If the state is WAITING_AUTH_I:

There are a number of reasons that you may receive an Identity Message in this state. Perhaps your correspondent simply started a new DAKE or they resent their Identity Message. On some networks, like AIM, if your correspondent is logged in multiple times, each of his clients will send an Identity Message in response to a Query Message. Resending the same Auth-R Message in response to each of those messages will prevent compounded confusion, since each of their clients will see each of the Auth-R Messages you send.

- Validate the Identity message. Ignore the message if validation fails.
- If validation succeeds:
    - Forget the old their_ecdh, their_dh, their_ecdh_first, their_dh_first and Client Profile from the previously received Identity message.
    - Send a new Auth-R message with the new values received.

If the state is ENCRYPTED_MESSAGES or FINISHED:

- Validate the new Identity message. Ignore the message if validation fails.
- If validation succeeds:
    - Remember the sender's instance tag to use as the receiver's instance tag for future messages.
    - Reply with an Auth-R message.
    - Transition to the WAITING_AUTH_I state.

Otherwise: * Ignore the message.

Sending an Auth-R Message

- Generate and send an Auth-R Message.
- Transition to state WAITING_AUTH_I.

Receiving an Auth-R Message

If the state is WAITING_AUTH_R:

- If the receiver's instance tag in the message is not the sender's instance tag you are currently using, ignore the message.
- Validate the Auth-R message.
    - If validation fails:
        - Ignore the message.
        - Stay in state WAITING_AUTH_R.
    - If validation succeeds:
        - Reply with an Auth-I message, as defined in Sending an Auth-I Message section.
    - Transition to the ENCRYPTED_MESSAGES state.

If the state is not WAITING_AUTH_R:

- Ignore this message.

Sending an Auth-I Message

- Generate and send an Auth-I message.
- Initialize the double ratcheting, as defined in the Interactive DAKE Overview section.
- Transition to state ENCRYPTED_MESSAGES.

Receiving an Auth-I Message

- If the state is WAITING_AUTH_I:
    - If the receiver's instance tag in the message is not the sender's instance tag you are currently using, ignore this message.
    - Validate the Auth-I message.
    - If validation fails:
        - Ignore the message.
        - Stay in state WAITING_AUTH_I.
    - If validation succeeds:
        - Transition to state ENCRYPTED_MESSAGES.
        - Initialize the double ratcheting, as defined in the Interactive DAKE Overview section.
        - If a plaintext message is waiting to be sent, encrypt it and send it.

        - If there are stored received Data Messages, remove them from storage
        - there is no way these messages are valid for the current DAKE.

- If the state is not WAITING_AUTH_I:

    - Ignore this message.

Sending a Data Message to an offline participant

- Generate and send a Non-Interactive-Auth message (replace any state or key variables).
- Initialize the double ratcheting, as defined in the Non-Interactive DAKE Overview section.
- If not already in this state, transition ENCRYPTED_MESSAGES state.
- If there is a recent stored plaintext message, encrypt it and send it.

Receiving a Non-Interactive-Auth Message

- If the state is FINISHED or MSGSTATE_FINISHED:

    - Ignore the message.

- Else (including any interactive state):

    - If the receiver's instance tag in the message is not the sender's instance tag you are currently using:

    - Ignore this message.

    - Otherwise:

    - Forget any set values (state or key variables).
    - Validate the Non-Interactive-Auth message.
    - Initialize the double ratcheting, as defined in the Non-Interactive DAKE Overview section.
    - Transition to state to ENCRYPTED_MESSAGES.
    - If a plaintext message is waiting to be sent, encrypt it and send it.
    - If there are stored received Data Messages, remove them from storage
        - there is no way these messages are valid for the current DAKE.

Sending a Data Message

The ENCRYPTED_MESSAGES state is the only state where a participant is allowed to send encrypted data messages.

If the state is START, WAITING_AUTH_R, WAITING_AUTH_I:

- Queue the message for encrypting and sending it when the participant transitions to the ENCRYPTED_MESSAGES state.

If the state is FINISHED, the participant must start another OTRv4 conversation to send encrypted messages:

- Inform the user that the message cannot be sent at this time.
- Store the plaintext message for possible retransmission.

If the state is ENCRYPTED:

- Encrypt the message, and send it as a Data Message.
- Store any plaintext message for possible retransmission.

Receiving a Data Message

A received data message will look like this:

```
["?OTR" || protocol version || message type || sender's instance_tag ||receiver's instance tag ||
  flags || previous chain message number || ratchet id || message id || public ECDH key ||
  public DH key || enc(plaintext message || TLV) || authenticator ||
  old MAC keys to be revealed ]
```

If the version is 4:

- If the state is not ENCRYPTED_MESSAGES:

    - If this is the first DH ratchet after a DAKE, store this message for a configurable, short amount of time configurable by the client (10-60 minutes is recommended).
    - Otherwise:
    - Inform the user that an unreadable encrypted message was received by replying with an Error Message: ERROR_2.

- Otherwise:

    - Validate the data message:

        - Verify that the message type is 0x03.
        - Verify the MAC tag.
        - Check if the message version is allowed.
        - Check that the instance tag in the message is the instance tag you are currently using.
        - Verify that the public ECDH key is on curve Ed448. See Verifying that a point is on the curve section for details.
        - Verify that the public DH key is from the correct group. See Verifying that an integer is in the DH group section for details.

    - If the message is not valid in any of the above steps:

        - Inform the user that an unreadable encrypted message was received by replying with an Error Message: ERROR_1.

    - Otherwise:

        - Derive the corresponding decryption key depending if you are on a new DH ratchet, if you have stored keys or not. Try to decrypt the message.

        - If the message cannot be decrypted (e.g., this is a duplicated message which key has already been used) and the IGNORE_UNREADABLE flag is not set:

            - Inform the user that an unreadable encrypted message was received by replying with an Error Message: ERROR_1.

        - If the message cannot be decrypted and the IGNORE_UNREADABLE flag is set:

            - Ignore it instead of producing an error or a notification to the user.

        - If the message can be decrypted:

            - Display the human-readable part (if non empty) to the user. SMP TLVs should be addressed according to the SMP state machine.
            - If the received message contains a TLV type 1 (Disconnected):
            - Forget all encryption keys for this correspondent and transition the state to FINISHED.

        - If you have not sent a message to this correspondent in some (configurable) time, send a "heartbeat" message. The heartbeat message should have the IGNORE_UNREADABLE flag set.

If the version is 3:

- If msgstate is MSGSTATE_ENCRYPTED:

    - Verify the information (MAC, keyids, ctr value, etc) in the message.

    - If the instance tag in the message is not the instance tag you are currently using:

        - Discard the message and optionally warn the user.

    - If the verification succeeds:

        - Decrypt the message and display the human-readable part (if non-empty) to the user.
        - Update the D-H encryption keys, if necessary.
        - If you have not sent a message to this correspondent in some (configurable) time, send a "heartbeat" message, consisting of a Data Message encoding an empty plaintext. The heartbeat message should have the IGNORE_UNREADABLE flag set.
        - If the received message contains a TLV type 1, forget all encryption keys for this correspondent, and transition msgstate to MSGSTATE_FINISHED.

    - Otherwise, inform the user that an unreadable encrypted message was received, and reply with an Error Message, as defined in OTRv3 protocol.

- If msgstate is MSGSTATE_PLAINTEXT or MSGSTATE_FINISHED:

- Inform the user that an unreadable encrypted message was received, and reply with an Error Message, as defined in OTRv3 protocol.

Receiving an Error Message

- Detect if an error code exists in the form ERROR_x where x is a number.
- If the error code exists in the spec:
  - Display the human-readable error message to the user.
- Otherwise:
  - Ignore the message.

If using version 3 and ERROR_START_AKE policy is set (which expects that the AKE will start when receiving an OTR Error message, as defined in OTRv3):

- Reply with a Query Message.

User requests to end an OTRv4 Conversation

If state is START:

- Do nothing

If state is ENCRYPTED_MESSAGES:

- Send a data message with an encoding of the message with an empty human-readable part, and the TLV type 1.
- Transition to the START state.

If state is FINISHED:

- Transition to state START.

Socialist Millionaires Protocol (SMP)

The Socialist Millionaires Protocol allows two parties with secret information (x and y, respectively) to check whether ($x == y$) without revealing any additional information about the secrets.

OTRv4 makes a few changes to SMP:

- OTRv4 uses Ed448 as a cryptographic primitive. This changes the way values are serialized and how they are computed. To define the SMP values under Ed448, we reuse the previously defined generator G for Ed448:

```
G = (x=224580040295924300187604334099896036246789641632564134246125461686950415467406032909029192869357953282578032075146446173674602635247710,
y=298819210078481492676017930443930673437544040154080242095928241372331506189835876003536878655418784733982303233503462500531545062832660)
```

- OTRv4 creates fingerprints using SHAKE-256. The fingerprint is generated as:
  - $HWC(usage\_fingerprint \mid\mid byte(H) \mid\mid byte(F), 56)$
- SMP in OTRv4 uses all of the TLV Record Types as OTRv3, except for SMP Message 1Q. When SMP Message 1Q is used in OTRv4, SMP Message 1 is used in OTRv4. When a question is not present, the user specified question section has length 0 and value NULL. In OTRv3, SMP Message 1 is used when the user does not specify an SMP question. If a question is supplied, SMP Message 1Q is used.
- SMP in OTRv4 uses the same SMP State Machine as OTRv3, with the exception that SMPSTATE_EXPECT1 only accepts SMP Message 1. Note that this state machine has no effect on type 0 or type 1 TLVs, which are always allowed.
- While picking random values in Z_q for elliptic curve operations for SMP, take into account the Considerations while working with elliptic curve parameters section.

SMP Overview

The computations below use the SMP Secret Information.

Assuming that Alice begins the exchange:

Alice:

- Picks random values, each 57 bytes long, for $a2$ and $a3$ in Z_q.
- Picks random values, each 57 bytes long, for $r2$ and $r3$ in Z_q.
- Computes $c2 = \text{HashToScalar}(0x01 \mid\mid G * r2)$ and $d2 = r2 - a2 * c2$.
- Computes $c3 = \text{HashToScalar}(0x02 \mid\mid G * r3)$ and $d3 = r3 - a3 * c3$.
- Sends Bob a SMP message 1 with $G2a = G * a2$, $c2$, $d2$, $G3a = G * a3$, $c3$ and $d3$.

Bob:

- Validates that $G2a$ and $G3a$ are on the curve Ed448, that they are in the correct group and that they do not degenerate.
- Picks random values, each 57 bytes long, for $b2$ and $b3$ in Z_q.
- Picks random values, each 57 bytes long, for $r2$, $r3$, $r4$, $r5$ and $r6$ in Z_q.
- Computes $G2b = G * b2$ and $G3b = G * b3$.
- Computes $c2 = \text{HashToScalar}(0x03 \mid\mid G * r2)$ and $d2 = r2 - b2 * c2$.
- Computes $c3 = \text{HashToScalar}(0x04 \mid\mid G * r3)$ and $d3 = r3 - b3 * c3$.
- Computes $G2 = G2a * b2$ and $G3 = G3a * b3$.
- Computes $Pb = G3 * r4$ and $Qb = G * r4 + G2 * (y \bmod q)$, where y is the SMP secret value.
- Computes $cp = \text{HashToScalar}(5 \mid\mid G3 * r5 \mid\mid G * r5 + G2 * r6)$, $d5 = r5 - r4 * cp$ and $d6 = (r6 - (y \bmod q) * cp) \bmod q$.
- Sends Alice a SMP message 2 with $G2b$, $c2$, $d2$, $G3b$, $c3$, $d3$, Pb, Qb, cp, $d5$ and $d6$.

Alice:

- Validates that $G2b$ and $G3b$ are on the curve Ed448, that they are in the correct group and that they do not degenerate.
- Computes $G2 = G2b * a2$ and $G3 = G3b * a3$.
- Picks random values, each 57 bytes long, for $r4$, $r5$, $r6$ and $r7$ in Z_q.
- Computes $Pa = G3 * r4$ and $Qa = G * r4 + G2 * (x \bmod q)$, where x is the SMP secret value.
- Computes $cp = \text{HashToScalar}(0x06 \mid\mid G3 * r5 \mid\mid G * r5 + G2 * r6)$, $d5 = r5 - r4 * cp$ and $(d6 = r6 - (x \bmod q) * cp) \bmod q$.
- Computes $Ra = (Qa - Qb) * a3$.
- Computes $cr = \text{HashToScalar}(0x07 \mid\mid G * r7 \mid\mid (Qa - Qb) * r7)$ and $d7 = r7 - a3 * cr$.
- Sends Bob a SMP message 3 with Pa, Qa, cp, $d5$, $d6$, Ra, cr and $d7$.

Bob:

- Validates that Pa, Qa, and Ra are on the curve Ed448 that they are in the correct group and that they do not degenerate.
- Picks a random value of 57 bytes long for $r7$ in Z_q.
- Computes $Rb = (Qa - Qb) * b3$.
- Computes $Rab = Ra * b3$.
- Computes $cr = \text{HashToScalar}(0x08 \mid\mid G * r7 \mid\mid (Qa - Qb) * r7)$ and $d7 = r7 - b3 * cr$.
- Checks whether $Rab == Pa - Pb$.
- Sends Alice a SMP message 4 with Rb, cr, $d7$.

Alice:

- Validates that Rb is on curve Ed448. See Verifying that a point is on the curve section for details.
- Computes $Rab = Rb * a3$.
- Checks whether $Rab == Pa - Pb$.

If everything is done correctly, then Rab should hold the value of $(Pa - Pb) * ((G2 * a3 * b3) * (x - y))$. This test will only succeed if the secret information provided by each participant are equal (essentially $x == y$). Further, since $G2 * a3 * b3$ is a random number not known to any party, if x is not equal to y, no other information is revealed.

Secret Information

The secret information x and y compared during this protocol contains not only information entered by the users, but also information unique to the conversation in which SMP takes place. This includes the Secure Session ID (SSID) whose creation is described here and here.

The format for the secret information is:

```
Version (BYTE)
  The version of SMP used. The version described here is 1.
Initiator fingerprint (56 BYTEs)
  The fingerprint that the party initiating SMP is using in the current
  conversation.
Responder fingerprint (56 BYTEs)
  The fingerprint that the party that did not initiate SMP is using in the
  current conversation.
Secure Session ID or SSID (8 BYTEs)
User-specified secret (DATA)
  The input string given by the user at runtime.
  It is encoded as UTF-8.
```

The first 57 bytes of a SHAKE-256 hash of the above is taken, the digest is then pruned as defined in the Considerations while working with elliptic curve parameters section. This digest then becomes the SMP secret value (x or y) to be used in SMP. The additional fields ensure that not only do both parties know the same secret input string, but no man-in-the-middle is capable of reading their communication either:

```
x or y = HWC(usage_SMP_secret || version || Initiator fingerprint ||
Responder fingerprint || Secure Session ID or SSID || User-specified secret),
57)
```

SMP Hash Function

There are many places where the 57 bytes of a SHAKE-256 hash are taken of an integer followed by other values. This is defined as $\text{HashToScalar}(i \mid\mid v)$ where i is an integer used to distinguish the calls to the hash function and v are some values. Hashing is done in this way to prevent Alice from replaying Bob's zero knowledge proofs or vice versa.

SMP Message 1

Alice sends SMP message 1 to begin an ECDH exchange to determine two new generators, $g2$ and $g3$. A valid SMP message 1 is generated as follows:

1. Determine her secret input x, which is to be compared to Bob's secret y, as specified in the Secret Information section.
2. Pick random values, each 57 bytes long, for $a2$ and $a3$ in Z_q. These will be Alice's exponents for the ECDH exchange to pick generators. These random values should be hashed and pruned as defined in the Considerations while working with elliptic curve parameters section prior to be used.
3. Pick random values of 57-bytes long for $r2$ and $r3$ in Z_q. These will be used to generate zero-knowledge proofs that this message was created according to the SMP protocol. These random values should be hashed and pruned as defined in the Considerations while working with elliptic curve parameters section prior to be used.
4. Interpret $a2$, $a3$, $r2$ and $r3$ as little-endian integers forming scalars.
5. Compute $G2a = G * a2$ and $G3a = G * a3$. Check that $G2a$ and $G3a$ are on curve Ed448. See Verifying that a point is on the curve section for details.
6. Generate a zero-knowledge proof that the value $a2$ is known by setting $c2 = \text{HashToScalar}(0x01 \mid\mid G * r2)$ and $d2 = r2 - a2 * c2 \bmod q$.
7. Generate a zero-knowledge proof that the value $a3$ is known by setting $c3 = \text{HashToScalar}(0x02 \mid\mid G * r3)$ and $d3 = r3 - a3 * c3 \bmod q$.
8. Store the values of x, $a2$ and $a3$ for use later in the protocol.

The SMP message 1 has the following data and format:

```
Question (DATA)
  A user-specified question, which is associated with the user-specified secret
  information. If there is no question input from the user, the length of this
  is 0 and the data is 'NULL'.

G2a (POINT)
  Alice's half of the ECDH exchange to determine G2.

c2 (SCALAR), d2 (SCALAR)
  A zero-knowledge proof that Alice knows the value associated with her
  transmitted value G2a.

G3a (POINT)
  Alice's half of the ECDH exchange to determine G3.

c3 (SCALAR), d3 (SCALAR)
  A zero-knowledge proof that Alice knows the value associated with her
  transmitted value G3a.
```

SMP Message 2

SMP message 2 is sent by Bob to complete the ECDH exchange to determine the new generators, $g2$ and $g3$. It also begins the construction of the values used in the final comparison of the protocol. A valid SMP message 2 is generated as follows:

1. Validate that $G2a$ and $G3a$ are on curve Ed448. See Verifying that a point is on the curve section for details.
2. Determine Bob's secret input y, which is to be compared to Alice's secret x.
3. Pick random values, each 57 bytes long, for $b2$ and $b3$ in Z_q. These will be used for creating the generators $g2$ and $g3$. These random values should be hashed and pruned as defined in the Considerations while working with elliptic curve parameters section prior to be used.
4. Pick random values, each 57 bytes long for $r2$, $r3$, $r4$, $r5$ and $r6$ in Z_q. These will be used to add a blinding factor to the final results, and to generate zero-knowledge proofs that state that this message was created honestly. These random values should be hashed and pruned as defined in the Considerations while working with elliptic curve parameters section prior to be used.
5. Interpret $b2$, $b3$, $r2$ and $r3$, $r4$, $r5$ and $r6$ as little-endian integers forming scalars.
6. Compute $G2b = G * b2$ and $G3b = G * b3$. Check that $G2b$ and $G3b$ are on curve Ed448. See Verifying that a point is on the curve section for details.
7. Generate a zero-knowledge proof that the value $b2$ is known by setting $c2 = HashToScalar(0x03 || G * r2)$ and $d2 = r2 - b2 * c2 \bmod q$.
8. Generate a zero-knowledge proof that the value $b3$ is known by setting $c3 = HashToScalar(0x04 || G * r3)$ and $d3 = r3 - b3 * c3 \bmod q$.
9. Compute $G2 = G2a * b2$ and $G3 = G3a * b3$. Check that $G2$ and $G3$ are on curve Ed448. See Verifying that a point is on the curve section for details.
10. Interpret y as little-endian integer forming a scalar.
11. Compute $Pb = G3 * r4$ and $Qb = G * r4 + G2 * y$. Check that Pb and Qb are on curve Ed448. See Verifying that a point is on the curve section for details.
12. Generate a zero-knowledge proof that Pb and Qb were created according to the protocol by setting $cp = HashToScalar(0x05 || G3 * r5 || G * r5 + G2 * r6)$, $d5 = r5 - r4 * cp \bmod q$ and $d6 = (r6 - y * cp) \bmod q$.
13. Store the values of $G3a$, $G2$, $G3$, $b3$, Pb and Qb for use later in the protocol.

The SMP message 2 has the following data and format:

```
G2b (POINT)
  Bob's half of the DH exchange to determine G2.

c2 (SCALAR), d2 (SCALAR)
  A zero-knowledge proof that Bob knows the exponent associated with his
  transmitted value G2b.

G3b (POINT)
  Bob's half of the ECDH exchange to determine G3.

c3 (SCALAR), d3 (SCALAR)
  A zero-knowledge proof that Bob knows the exponent associated with his
  transmitted value G3b.

Pb (POINT), Qb (POINT)
  These values are used in the final comparison to determine if Alice and Bob
  share the same secret.

cp (SCALAR), d5 (SCALAR), d6 (SCALAR)
  A zero-knowledge proof that Pb and Qb were created according to the protocol
  given above.
```

SMP Message 3

SMP message 3 is Alice's final message in the SMP exchange. It has the last of the information required by Bob to determine if $x == y$. A valid SMP message 3 is generated as follows:

1. Validate that $G2b$, $G3b$, Pb, and Qb are on curve Ed448. See Verifying that a point is on the curve section for details.
2. Pick random values, each 57 bytes long, for $r4$, $r5$, $r6$ and $r7$ in Z_q. These will be used to add a blinding factor to the final results and to generate zero-knowledge proofs that this message was created honestly. These random values should be hashed and pruned as defined in the Considerations while working with elliptic curve parameters section prior to be used.
3. Interpret $r4$, $r5$ and $r6$ as little-endian integers forming scalars.
4. Compute $G2 = G2b * a2$ and $G3 = G3b * a3$. Check that $G2$ and $G3$ are on curve Ed448. See Verifying that a point is on the curve section for details.
5. Interpret x as little-endian integer forming a scalar.
6. Compute $Pa = G3 * r4$ and $Qa = G * r4 + G2 * x$. Check that Pa and Qa are on curve Ed448. See Verifying that a point is on the curve section for details.
7. Generate a zero-knowledge proof that Pa and Qa were created according to the protocol by setting $cp = HashToScalar(0x06 || G3 * r5 || G * r5 + G2 * r6)$, $d5 = r5 - r4 * cp \bmod q$ and $d6 = ((r6 - x * cp) \bmod q$.
8. Compute $Ra = (Qa - Qb) * a3$. Check that Ra is on curve Ed448. See Verifying that a point is on the curve section for details.
9. Generate a zero-knowledge proof that Ra was created according to the protocol by setting $cr = HashToScalar(0x07 || G * r7 || (Qa - Qb) * r7)$ and $d7 = r7 - a3 * cr \bmod q$.

10. Store the values of G3b, Pa - Pb, Qa - Qb and a3 for use later in the protocol.

The SMP message 3 has the following data and format:

```
Pa (POINT), Qa (POINT)
  These values are used in the final comparison to determine if Alice and Bob
  share the same secret.

cp (SCALAR), d5 (SCALAR), d6 (SCALAR)
  A zero-knowledge proof that Pa and Qa were created according to the protocol
  given above.

Ra (POINT)
  This value is used in the final comparison to determine if Alice and Bob share
  the same secret.

cr (SCALAR), d7 (SCALAR)
  A zero-knowledge proof that Ra was created according to the protocol given
  above.
```

SMP Message 4

SMP message 4 is Bob's final message in the SMP exchange. It has the last of the information required by Alice to determine if x == y. A valid SMP message 4 is generated as follows:

1. Validate that Pa, Qa, and Ra are on curve Ed448. See Verifying that a point is on the curve section for details.
2. Pick a random value of 57-bytes long for r7 in Z_q. This will be used to generate Bob's final zero-knowledge proof that this message was created honestly. This random value should be hashed and pruned as defined in the Considerations while working with elliptic curve parameters section prior to be used.
3. Interpret r7 as little-endian integer forming a scalar.
4. Compute Rb = (Qa - Qb) * b3. Check that Rb is on curve Ed448. See Verifying that a point is on the curve section for details.
5. Generate a zero-knowledge proof that Rb was created according to the protocol by setting cr = HashToScalar(0x08 || G * r7 || (Qa - Qb) * r7) and d7 = r7 - b3 * cr mod q.

The SMP message 4 has the following data and format:

```
Rb (POINT)
  This value is used in the final comparison to determine if Alice and Bob
  share the same secret.

cr (SCALAR), d7 (SCALAR)
  A zero-knowledge proof that Rb was created according to this SMP protocol.
```

The SMP State Machine

OTRv4 does not change the state machine for SMP from OTRv3. But the following sections detail how values are computed differently during some states. Each case assumes that the protocol state is ENCRYPTED_MESSAGES. It must be taken into account that state SMPSTATE_EXPECT1 is reached whenever an error occurs or SMP is aborted. In that case, the protocol must be restarted from the beginning. Whenever the OTRv4 message state machine is in ENCRYPTED_MESSAGES state, the SMP state machine may progress. If at any point you are not in ENCRYPTED_MESSAGES, the SMP must abandon its state and return to its initial setup.

User requests to begin SMP

If smpstate is not set to SMPSTATE_EXPECT1:

- SMP is already underway. If you wish to restart the SMP, send a type 6 TLV (SMP abort) to the other party and then proceed as if smpstate was SMPSTATE_EXPECT1. Otherwise, you may simply continue the current SMP instance.

If smpstate is set to SMPSTATE_EXPECT1:

- No current exchange is underway. In this case, Alice creates a valid type 2 TLV (SMP message 1) as follows:
  1. Create a valid SMP Message 1 as defined in its section.
  2. Set smpstate to SMPSTATE_EXPECT2.

User requests to abort SMP

In all cases, send a type 6 TLV (SMP abort) to the correspondent and set smpstate to SMPSTATE_EXPECT1.

Receiving a SMP Message 1

If the instance tag in the message is not the instance tag you are currently using, ignore the message.

If smpstate is not SMPSTATE_EXPECT1:

- Set smpstate to SMPSTATE_EXPECT1
- Send a SMP abort to Alice.

If smpstate is SMPSTATE_EXPECT1:

- Verify Alice's zero-knowledge proofs for G2a and G3a:
  1. Check that both G2a and G3a are on curve Ed448. See Verifying that a point is on the curve section for details.
  2. Check that c2 == HashToScalar(0x01 || G * d2 + G2a * c2).
  3. Check that c3 == HashToScalar(0x02 || G * d3 + G3a * c3).
- Create a SMP message 2 and send it to Alice.
- Set smpstate to SMPSTATE_EXPECT3.

Receiving a SMP Message 2

If the instance tag in the message is not the instance tag you are currently using, ignore the message.

If smpstate is not SMPSTATE_EXPECT2:

- Set smpstate to SMPSTATE_EXPECT1 and send a SMP abort to Bob.

If smpstate is SMPSTATE_EXPECT2:

- Verify Bob's zero-knowledge proofs for G2b, G3b, Pb and Qb:
  1. Check that G2b, G3b, Pb and Qb are on curve Ed448. See Verifying that a point is on the curve section for details.
  2. Check that c2 == HashToScalar(0x03 || G * d2 + G2b * c2).
  3. Check that c3 == HashToScalar(0x04 || G * d3 + G3b * c3).
  4. Check that cp == HashToScalar(0x05 || G3 * d5 + Pb * cp || G * d5 + G2 * d6 + Qb * cp).
- Create a SMP message 3 and send it to Bob.
- Set smpstate to SMPSTATE_EXPECT4.

Receiving a SMP Message 3

If the instance tag in the message is not the instance tag you are currently using, ignore the message.

If smpstate is not SMPSTATE_EXPECT3:

- Set smpstate to SMPSTATE_EXPECT1 and send a SMP abort to Bob.

If smpstate is SMPSTATE_EXPECT3:

- Verify Alice's zero-knowledge proofs for Pa, Qa and Ra:
  1. Check that Pa, Qa and Ra are on curve Ed448. See Verifying that a point is on the curve section for details.
  2. Check that cp == HashToScalar(0x06 || G3 * d5 + Pa * cp || G * d5 + G2 * d6 + Qa * cp).
  3. Check that cr == HashToScalar(0x07 || G * d7 + G3a * cr || (Qa - Qb) * d7 + Ra * cr).
- Create a SMP message 4 and send it to Alice.
- Check whether the protocol was successful:
  1. Compute Rab = Ra * b3.
  2. Determine if x == y by checking the equivalent condition that Pa - Pb == Rab.
- Set smpstate to SMPSTATE_EXPECT1, as no more messages are expected from Alice.

Receiving a SMP Message 4

If the instance tag in the message is not the instance tag you are currently using, ignore the message.

If smpstate is not SMPSTATE_EXPECT4:

- Set smpstate to SMPSTATE_EXPECT1 and send a type 6 TLV (SMP abort) to Bob.

If smpstate is SMPSTATE_EXPECT4:

- Verify Bob's zero-knowledge proof for Rb:
  1. Check that `Rb` is on curve Ed448. See Verifying that a point is on the curve section for details.
  2. Check that `cr == HashToScalar(0x08 || G * d7 + G3b * cr || (Qa - Qb) * d7 + Rb * cr)`.
- Check whether the protocol was successful:
  1. `Compute Rab = Rb * a3`.
  2. Determine if `x == y` by checking the equivalent condition that `(Pa - Pb) == Rab`.
- Set smpstate to `SMPSTATE_EXPECT1`, as no more messages are expected from Bob.

Implementation Notes

Considerations for Networks that allow Multiple Clients

When using a transport network that allows multiple clients to be simultaneously logged in with the same peer identifier, make sure to identify the other participant by its client-specific identifier and not only the peer identifier (for example, using XMPP full JID instead of bare JID). Doing so allows establishing multiple OTR channels at the same time with multiple clients from the other participant. This can cost that the client manages this exposure (for example, XMPP clients can decide to reply only to the client you have more recently received a message from).

Forging Transcripts

OTRv4 expects each implementation of this specification to expose an interface for producing forged transcripts. These forging operations must use the same functions used for honest conversations. This section will outline the operations that must be exposed and include guidance to forge messages.

In OTRv4, anyone can forge messages after a conversation to make them look like they came from them. However, during a conversation, your correspondent is assured that the messages they see are authentic and unmodified. Easily forgeable transcripts achieve the offline deniability property: if someone claims a participant said something over OTR, they'll have no way to proof so, as anyone could have modify a transcript.

The major utilities for forging are:

```
Parse
  Parses OTRv4 messages to the values of each of the fields in them and shows
  these fields.

Modify Data Message
  If an encrypted data message cannot be read because you don't know the message
  key (or one of the chain keys used to derive this message key) but it can be
  guessed that the string 'x' appears at a given place in the message, a
  participant can replace that string with some new desired text with the same
  length. The result is a valid OTRv4 message that contains the new text. For
  example, if the string "hi" is accurately guessed to be at the beginning of
  an encrypted message, it can be replaced with the string "yo". Therefore, a
  valid data message can be created with new text.

  To achieve this:
  - XOR the old text and the new text. Store this value.
  - XOR the stored value again with the original encrypted message starting
    at a given offset.
  - Recalculate the MAC tag with the revealed MAC key associated with this
    message. The new tag is attached to the data message, replacing the old
    value.
```

Pseudocode for modifying data messages is included in the Appendices.

```
Read and Forge Data Message
  Read and forge allows someone in possession of a chain key to decrypt OTR
  messages or modify them as forgeries. It takes three inputs: the chain key,
  the OTRv4 message and a new plain text message (optional). If a new
  message is included, the original text is replaced with the new message and
  a new MAC tag is attached to the data message.

  To achieve this:
  - Decrypt the data message with the corresponding message key derived from
    the given chain key.
  - If a new message is given, replace the message with that one, encrypt it
    and create its mac accordingly.

Forge DAKE and Session Keys
  Any participant of an OTR conversation may forge a DAKE with another
```

participant as long as they have their Client Profile. This function will take the Client Profile and the secret long-term key of one participant, and the Client Profile of the other (or Prekey Ensemble in case of non-interactive DAKE). It will return a DAKE transcript between the two parties. The participant's private key is required since it is used to authenticate the key exchange, but the resulting transcript is created in such a way that a cryptographic expert cannot identify which client profile owner authenticated the conversation.

Show MAC Key
This function takes an message key and shows the corresponding MAC key. 'Show MAC key' may be used with the ReMAC Message function below in the case where a message key has been compromised by an attacker who wishes to forge messages.

ReMAC Message
This will make a new OTR Data Message with a given MAC key and an original OTR data message. The user's message in the OTR data message is already encrypted. A new MAC tag will be generated and replaced for the message. An attacker may use this function to forge messages with a compromised MAC key.

Impersonate Initiator or Responder
In the case of wanting to impersonate the responder, this function takes the long-term secret key of the Identifier as input. It will make the owner of this long-term secret key to pretend to be the Responder by executing the RSig functionality with those keys. This can happen in the interactive and non-interactive DAKE.
In the case of wanting to impersonate the identifier, this function takes the long-term secret key of the Responder as input. It will make the owner of this long-term secret key to pretend to be the Identifier by executing the RSig functionality with those keys. This can only happen in the interactive DAKE.

False Prekey Ensemble
This function will return a false Prekey Ensemble for the Identifier. It will only be used for the non-interactive DAKE.

Forge Entire Transcript
The Forge Entire Transcript function will allow one participant to completely forge a transcript between them and another person in a way that its forgery cannot be cryptographically proven. The input will be: one participant's Client Profile, their secret key, another participant's Client Profile, and a list of plain text messages corresponding to what messages were exchanged. Each message in the list will have the structure: 1) sender 2) plain text message, so that the function may precisely create the desired transcript. The participant's private key is required since it is used to authenticate the key exchange, but the resulting transcript is created in such a way that a cryptographic expert cannot identify which Client Profile owner authenticated the conversation.

Licensing and Use

The OTR team does not review implementations or specify which ones are compliant are not. Software implementors are free to implement this specification in any way they choose - under limitations of software licenses if using existing software.

Appendices

Ring Signature Authentication

The Authentication scheme consists of two functions:

- An authentication function: `sigma = RSig(A1, a1, {A1, A2, A3}, m)`.
- A verification function: `RVrf({A1, A2, A3}, sigma, m)`.

Domain Parameters

We reuse the previously defined G generator in elliptic curve parameters:

```
G = (x=22458004029592430018760433409989603624678964163256413424612546168695
     04154674060329090291928693579532825780320751464461736746026352477 10,
   y=29881921007848149267601793044393067343754404015408024209592824137233
```

```
150618983587600353687865541878473398230323350346250053154506283266O)
```

Authentication: RSig(A1, a1, {A1, A2, A3}, m):

`RSig` produces a SoK (signature of knowledge), named `sigma`, bound to the message `m`, that demonstrates knowledge of a private key corresponding to one of three public keys.

In the case the DAKEs used for interactive and non-interactive, `A1` is the public value associated with `a1`, that is, `A1 = G * a1` and `m` is the message to authenticate.

To compute `RSig`, without loss of generality:

`A1`, `A2`, and `A3` should be checked to verify that they are on the curve Ed448. See Verifying that a point is on the curve section for details.

1. Pick random values `t1, c2, c3, r2, r3` in `Z_q`. These random values should be hashed and pruned as defined in the Considerations while working with elliptic curve parameters section prior to be used.
2. Compute `T1 = G * t1`.
3. Compute `T2 = G * r2 + A2 * c2`.
4. Compute `T3 = G * r3 + A3 * c3`.
5. Compute `c = HashToScalar(usage_auth || G || q || A1 || A2 || A3 || T1 || T2 || T3 || m)`.
6. Compute `c1 = c - c2 - c3 (mod q)`.
7. Compute `r1 = t1 - c1 * a1 (mod q)`. Securely delete `t1`.
8. Send `sigma = (c1, r1, c2, r2, c3, r3)`.

This function can be generalized so it is not possible to determine which secret key was used to produce this ring signature, even if all secret keys are revealed. For this, constant-time conditional operations should be used.

The prover knows a secret `ai` and, therefore:

1. Pick random values `t1, t2, t3, c1, c2, c3, r1, r2, r3` in `Z_q`. These random values should be hashed and pruned as defined in the Considerations while working with elliptic curve parameters section prior to be used.
2. Compute:

```
P = G * ai
eq1 = constant_time_eq(P, A1)
eq2 = constant_time_eq(P, A2)
eq3 = constant_time_eq(P, A3)
```

1. Depending of the result of the above operations, compute:

```
T1 = constant_time_select(eq1, encode(G * t1), encode(G * r1 + A1 * c1))
T2 = constant_time_select(eq2, encode(G * t2), encode(G * r2 + A2 * c2))
T3 = constant_time_select(eq3, encode(G * t3), encode(G * r3 + A3 * c3))
```

1. Compute `c = HashToScalar(usage_auth || G || q || A1 || A2 || A3 || T1 || T2 || T3 || m)`.
2. For whichever equally returns true (if `eqi == 1`, `eqj == 0` and `eqk == 0`, for `i != j != k`): `ci = c - cj - ck (mod q)`.
3. For whichever equally returns true (for example, if `eqi == 1`): `ri = ti - ci * ai (mod q)`. Securely delete `ti`.
4. Compute `sigma = (ci, ri, cj, rj, ck, rk)`.

If the prover knows `a2`, for example, the `RSig` function looks like this: `RSig(A2, a2, {A1, A2, A3}, m)`

1. Pick random values `t2, c1, c3, r1, r3` in `Z_q`. These random values should be hashed and pruned as defined in the Considerations while working with elliptic curve parameters section prior to be used.
2. Compute `T2 = G * t2`.
3. Compute `T1 = G * r1 + A1 * c1`.
4. Compute `T3 = G * r3 + A3 * c3`.
5. Compute `c = HashToScalar(usage_auth || G || q || A1 || A2 || A3 || T1 || T2 || T3 || m)`.
6. Compute `c2 = c - c1 - c3 (mod q)`.
7. Compute `r2 = t2 - c2 * a2 (mod q)`.
8. Send `sigma = (c1, r1, c2, r2, c3, r3)`.

The order of elements passed to `H` and sent to the verifier must not depend on the secret known by the prover (otherwise, the key used to produce the proof can be inferred in practice).

Verification: RVrf({A1, A2, A3}, sigma, m)

RVrf is the verification function for the SoK sigma, created by RSig.

A1, A2, and A3 should be checked to verify that they are on curve Ed448.

1. Parse sigma to retrieve components (c1, r1, c2, r2, c3, r3).
2. Compute T1 = G * r1 + A1 * c1
3. Compute T2 = G * r2 + A2 * c2
4. Compute T3 = G * r3 + A3 * c3
5. Compute h = HWC(usage_auth || G || q || A1 || A2 || A3 || T1 || T2 || T3 || m).
6. Compute c = h (mod q).
7. Check if $c \stackrel{?}{=} c1 + c2 + c3 \pmod{q}$. If it is true, verification succeeds. If not, it fails.

HashToScalar

This function is HashToScalar(usageID || d, 57), where d is an array of bytes.

1. Compute h = HWC(usageID || d, 57).
2. Interpret the buffer as a little-endian integer, forming a scalar. Return this scalar.

Modify an Encrypted Data Message

In this example, a forger guesses that "hi" is at the beginning of an encrypted message. Thus, its offset is 0. The forger wants to replace "hi" with "yo".

```
offset = 0
old_text = "hi"
new_text = "yo"
text_length = string_length_of(old_text)
old_encrypted_message = get_from_data_message()
encrypted_message_length = string_length_of(old_encrypted_message)

for (i=0; i < text_length && offset+i < encrypted_message_length; i++) {
   old_encrypted_message[offset+i] ^= old_text[i] ^ new_text[i]
}

new_encrypted_message = old_encrypted_message

new_mac_tag = mac(new_encrypted_message, revealed_mac_key)

new_data_message = replace(old_data_message, new_encrypted_message, new_mac_tag)
```

OTRv3 Specific Encoded Messages

D-H Commit Message

This is the first message of OTRv3 AKE. Bob sends it to Alice to commit to a choice of D-H encryption key (but the key itself is not yet revealed). This allows the secure session id to be much shorter than in OTRv1, while still preventing a man-in-the-middle attack on it.

The D-H Commit Message consists of the protocol version, the message type, the sender's instance tag, the receiver's instance tag, the encoded encrypted sender's public key and the hashed sender's public key.

D-H Key Message

This is the second message of OTRv3 AKE. Alice sends it to Bob.

It consists of: the protocol version, the message type, the sender's instance tag, the receiver's instance tag and the public key.

Reveal Signature Message

This is the third message of the OTRv3 AKE. Bob sends it to Alice, revealing his D-H encryption key (and thus opening an encrypted channel), and also authenticating himself (and the parameters of the channel, preventing a man-in-the-middle attack on the channel itself) to Alice.

It consists of: the protocol version, the message type, the sender's instance tag, the receiver's instance tag, the revealed key, the encrypted signature and the MAC of the signature.

Signature Message

This is the final message of the OTRv3 AKE. Alice sends it to Bob, authenticating herself and the channel parameters to him.

It consists of: the protocol version, the message type, the sender's instance tag, the receiver's instance tag, the encrypted signature and the MAC of the signature.

Data Message

In OTRv3, this message is used to transmit a private message to the correspondent. It is also used to reveal old MAC keys.

Receiving a D-H Commit Message

If the message is version 3 and version 3 is not allowed:

- Ignore the message.

Otherwise:

If authstate is AUTHSTATE_NONE:

- Reply with a D-H Key Message, and transition authstate to AUTHSTATE_AWAITING_REVEALSIG.

If authstate is AUTHSTATE_AWAITING_DHKEY:

- This indicates that you have already sent a D-H Commit message to your peer, but that it either didn't receive it, or just didn't receive it yet and has sent you one as well. The symmetry will be broken by comparing the hashed g^x you sent in your D-H Commit Message with the one you received, considered as 32-byte unsigned big-endian values.
- If yours is the higher hash value:
    - Ignore the incoming D-H Commit message, but resend your D-H Commit message.
- Otherwise:
    - Forget the old encrypted g^x value that you sent earlier, and pretend you're in AUTHSTATE_NONE. For example, reply with a D-H Key Message, and transition authstate to AUTHSTATE_AWAITING_REVEALSIG.

If authstate is AUTHSTATE_AWAITING_REVEALSIG:

- Retransmit your D-H Key Message (the same one you sent when you entered AUTHSTATE_AWAITING_REVEALSIG). Forget the old D-H Commit message and use this new one instead.

  There are a number of reasons this might happen, including:

    - Your correspondent simply started a new AKE.
    - Your correspondent resent his D-H Commit message, as specified above.
    - On some networks, like AIM, if your correspondent is logged in multiple times, each of his clients will send a D-H Commit Message in response to a Query Message. Resending the same D-H Key Message in response to each of those messages will prevent confusion, since each of the clients will see each of the D-H Key Messages sent.

If authstate is AUTHSTATE_AWAITING_SIG:

- Reply with a new D-H Key message and transition authstate to AUTHSTATE_AWAITING_REVEALSIG.

Receiving a D-H Key Message

If the instance tag in the message is not the instance tag you are currently using:

- Ignore the message.

If the message is version 3 and version 3 is not allowed:

- Ignore this message.

Otherwise:

If authstate is AUTHSTATE_AWAITING_DHKEY:

- Reply with a Reveal Signature Message and transition authstate to AUTHSTATE_AWAITING_SIG.

If authstate is AUTHSTATE_AWAITING_SIG:

- If this D-H Key message is the same you received earlier (when you entered AUTHSTATE_AWAITING_SIG):
    - Retransmit your Reveal Signature Message.
- Otherwise:
    - Ignore the message.

If authstate is AUTHSTATE_NONE, AUTHSTATE_AWAITING_REVEALSIG, or AUTHSTATE_V1_SETUP:

- Ignore the message.

Receiving a Reveal Signature Message

If the instance tag in the message is not the instance tag you are currently using, ignore the message.

If version 3 is not allowed:

- Ignore this message.

Otherwise:

If authstate is AUTHSTATE_AWAITING_REVEALSIG:

- Use the received value of `r` to decrypt the value of `g^x` received in the D-H Commit Message, and verify the hash therein.
- Decrypt the encrypted signature, and verify the signature and the MACs. If everything checks out:
    - Reply with a Signature Message.
    - Transition authstate to AUTHSTATE_NONE.
    - Transition msgstate to MSGSTATE_ENCRYPTED.
    - If there is a recent stored message, encrypt it and send it as a Data Message.
- Otherwise:
    - Ignore the message.

If authstate is AUTHSTATE_NONE, AUTHSTATE_AWAITING_DHKEY or AUTHSTATE_AWAITING_SIG:

- Ignore the message.

Receiving a Signature Message

If the instance tag in the message is not the instance tag you are currently using:

- Ignore the message.

If version 3 is not allowed:

- Ignore this message.

Otherwise:

If authstate is AUTHSTATE_AWAITING_SIG:

- Decrypt the encrypted signature, and verify the signature and the MACs. If everything checks out:
    - Transition authstate to AUTHSTATE_NONE.
    - Transition msgstate to MSGSTATE_ENCRYPTED.
    - If there is a recent stored message, encrypt it and send it as a Data Message.
- Otherwise, ignore the message.

If authstate is AUTHSTATE_NONE, AUTHSTATE_AWAITING_DHKEY or AUTHSTATE_AWAITING_REVEALSIG:

- Ignore the message.

OTRv3 Protocol State Machine

OTRv3 defines three main state variables:

Message State

The message state variable `msgstate` controls what happens to outgoing messages typed by the user. It can take one of three values:

```
MSGSTATE_PLAINTEXT
  This state indicates that outgoing messages are sent without encryption. This
  is the state used before an OTRv3 conversation is initiated. This is the
  initial state, and the only way to subsequently enter this state is for the
  user to explicitly request so via some UI operation.

MSGSTATE_ENCRYPTED
  This state indicates that outgoing messages are sent encrypted. This is the
  state that is used during an OTRv3 conversation. The only way to enter this
  state is when the authentication state machine (below) is completed.

MSGSTATE_FINISHED
  This state indicates that outgoing messages are not delivered at all. This
  state is entered only when the other party indicates that its side of the
  conversation has ended. For example, if Alice and Bob are having an OTR
  conversation, and Bob instructs his OTR client to end its private session with
  Alice (for example, by logging out), Alice will be notified of this, and her
  client will switch to 'MSGSTATE_FINISHED' mode. This prevents Alice from
  accidentally sending a message to Bob in plaintext (consider what happens if
  Alice was in the middle of typing a private message to Bob when he suddenly
  logs out, just as Alice hits Enter.)
```

Authentication State

The authentication state variable `authstate` can take one of four values:

```
AUTHSTATE_NONE
  This state indicates that the authentication protocol is not currently in
  progress. This is the initial state.

AUTHSTATE_AWAITING_DHKEY
  After Bob initiates the authentication protocol by sending Alice the 'D-H
  Commit Message', he enters this state to await Alice's reply.

AUTHSTATE_AWAITING_REVEALSIG
  After Alice receives Bob's D-H Commit Message, and replies with her own 'D-H
  Key Message', she enters this state to await Bob's reply.

AUTHSTATE_AWAITING_SIG
  After Bob receives Alice's 'D-H Key Message', and replies with his own Reveal
  Signature Message, he enters this state to await Alice's reply.
```

Elliptic Curve Operations

Point Addition

For point addition, the following method is recommended, as defined in RFC 8032. A point (x,y) is represented in projective coordinates (X, Y, Z), with $x = X/Z$, $y = Y/Z$ with $Z != 0$.

The neutral point is $(0,1)$, or equivalently in projective coordinates $(0, Z, Z)$ for any non-zero Z.

The following formula is for adding two points, $(x3,y3) = (x1,y1) + (x2,y2)$ (or $X_1 : Y_1 : Z_1) + X_2 : Y_2 : Z_2 = X_3 : Y_3 : Z_3$) on untwisted Edwards curve (i.e., $a = 1$) with non-square d, as defined in [10]. They are complete (they work for any pair of valid input points):

Compute:

```
A = Z1 * Z2
B = A^2
C = X1 * X2
D = Y1 * Y2
E = d * C * D
F = B - E
G = B + E
H = (X1 + Y1) * (X2 + Y2)
X3 = A * F * (H - C - D)
Y3 = A * G * (D - C)
```

```
Z3 = F * G
```

References

1. Goldberg, I. and Unger, N. (2016). Improved Strongly Deniable Authenticated Key Exchanges for Secure Messaging, Waterloo, Canada: University of Waterloo. Available at: http://cacr.uwaterloo.ca/techreports/2016/cacr2016-06.pdf
2. Perrin, T. and Marlinspike, M. (2016). The Double Ratchet Algorithm. [online]signal.org. Available at: https://whispersystems.org/docs/specifications/doubleratchet
3. Bernstein, D. (2008). ChaCha, a variant of Salsa20, Chicago, USA: The University of Illinois at Chicago. Available at: https://cr.yp.to/chacha/chacha-20080128.pdf
4. Hamburg, M. (2015). Ed448-Goldilocks, a new elliptic curve, NIST ECC workshop. Available at: https://eprint.iacr.org/2015/625.pdf
5. Hamburg, M., Langley, A. and Turner, S. (2016). Elliptic Curves for Security, Internet Engineering Task Force, RFC 7748. Available at: http://www.ietf.org/rfc/rfc7748.txt
6. Kojo, M. (2003). More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE), Internet Engineering Task Force, RFC 3526. Available at: https://www.ietf.org/rfc/rfc3526.txt
7. Off-the-Record Messaging Protocol version 3. Available at: https://otr.cypherpunks.ca/Protocol-v3-4.1.1.html
8. Meijer, R., Millard, P. and Saint-Andre, P. (2017). XEP-0060: Publish-Subscribe Available at: https://xmpp.org/extensions/xep-0060.pdf
9. Josefsson, S. and Liusvaara, I. (2017). Edwards-curve Digital Signature Algorithm (EdDSA), Internet Engineering Task Force, RFC 8032. Available at: https://tools.ietf.org/html/rfc8032
10. Bernstein, D. and T. Lange. (2007). Projective coordinates for Edwards curves, The 'add-2007-bl' addition formulas. Available at: http://www.hyperelliptic.org/EFD/g1p/auto-edwards-projective.html#addition-add-2007-bl
11. Blake-Wilson, S., Johnson, D. and Menezes, A. (1997) Key Agreement Protocols and their Security Analysis. Available at: https://dl.acm.org/citation.cfm?id=742138
12. Gunn, L. J., Vieitez Parra, R. and Asokan, N. (2018) On The Use of Remote Attestation to Break and Repair Deniability. Available at: https://eprint.iacr.org/2018/424.pdf
13. Unger, N. and Goldberg, I. (2015). Deniable Key Exchanges for Secure Messaging. Available at: https://www.cypherpunks.ca/~iang/pubs/dake-ccs15.pdf
14. Antipa, A., Brown D., Menezes, A., Struik R., and Vanstone, S. (2015). Validation of Elliptic Curve Public Keys. Available at: https://iacr.org/archive/pkc2003/25670211/25670211.pdf
15. Bernstein, D., Hamburg, M., Krasnova, A., and Lange T. (2013). Elligator: Elliptic-curve points indistinguishable from uniform random strings. Available at: https://elligator.cr.yp.to/elligator-20130828.pdf
16. Nir, Y. and Langley, A. (2015). ChaCha20 and Poly1305 for IETF Protocols, Internet Research Task Force (IRTF), RFC 7539. Available at: https://tools.ietf.org/html/rfc7539